REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
12-07-2004	Final Report	(3/7/02-12/31/04)

4. TITLE AND SUBTITLE
Survivability Extensions for Dynamic UltraLog Environments

5a. CONTRACT NUMBER
MDA972-02-C-0025

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Marc Brittan

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The Boeing Company,
P.O. Box 3707, M/C 7L-20,
Seattle, WA 98124-2207

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Dr. Mark Greaves
Defense Advanced Research
Projects Agency (DARPA)

3701 N. Fairfax Drive
Arlington, VA 22203-1714

10. SPONSOR/MONITOR'S ACRONYM(S)
IXO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
CDRL A004

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This final report describes the status of work performed during the contr
period of performance from March 2002 to December 2004 on the DARPA proje
titled "Survivability Extensions for Dynamic UltraLog Environments". The
objective of this effort was to:

1. Research and demonstrate new algorithms for improving metrics of performance and risk of a distributed agent system. This is done by building a mathematical model of the system, and using optimization techniques
to improve the performance and risk profiles.

2. Integrate these algorithms into the fully operational Ultralog agent society. The tools for managing the state of the distributed agent system should themselves fit seamlessly into the existing Ultralog architecture.

15. SUBJECT TERMS
Optimization, Distributed Systems, Agents, Performance, Risk

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Marc Brittan
Unclassified	Unclassified	Unclassified	SAR	218	19b. TELEPHONE NUMBER *(include area code)* 425-865-2182

MDA972-02-C-0025 Final Report

Survivability Extensions for Dynamic UltraLog Environments

This material is based upon work supported by the
Defense Advanced Research Projects Agency
Advanced Technology Office
ARPA Order No. N047/00/Program Code 2G10
Survivability Extensions for Dynamic UltraLog Environments
Issued by DARPA/CMO under Contract #MDA972-02-C-0025

Marc Brittan
The Boeing Company
Phantom Works
Mathematics and Computing Technology
P.O. Box 3707 M/C 7L-20
Seattle, WA 98124-2207

Survivability Extensions for Dynamic UltraLog Environments

This material is based upon work supported by the
Defense Advanced Research Projects Agency
Advanced Technology Office
ARPA Order No. N047/00/Program Code 2G10
Survivability Extensions for Dynamic UltraLog Environments
Issued by DARPA/CMO under Contract #MDA972-02-C-0025

Contents

Forward

This Final Report, Prepared by the Boeing Company Mathematics and Computing Technology organization, is provided under DARPA contract MDA972-02-C-0025 for research entitled “Survivability Extensions for Dynamic UltraLog Environments”. The performance period for this effort is from March 2002 through December 2004.

1 Task Objectives

Our objectives in Ultralog were to provide improved state management for the Ultralog system of mobile agents. The primary focus of our research was to develop techniques and working software that would allow the system to dynamically reconfigure itself, either after attack or on an ad-hoc basis, in such a way that the system exhibited improved performance and reduced risk. Since the system is designed around mobile agents, our task was to create a “Load Balancer” that would assign agents to servers in a near-optimal way based on state risk and state performance.

A state of the system is defined as an assignment of agents to servers on a network, and the associated routings required to support inter-agent communications. A state is “healthy” or effective if all of the servers are within the healthy zone of operation (e.g., below 80% CPU and memory utilization, no significant paging or queuing at processors or other resources), and if the overall communications requirements between servers on the net falls below the bandwidths of the data links supporting that state. If a state is “healthy”, it still may have excessively long run times for major processes of interest in the state, since it may have CPU intensive tasks timeshared on slow processors. The assignment of tasks to resources is a major component of distributed system design. It is also a hard problem in mathematics, which complicates the job of designing a system, and of course managing a system in real time.

To accomplish our stated objective, the project was divided into a base year, where we used our windows based solver to demonstrate our solver in the Ultralog agent society, followed by two option years, where we proceeded to build a tightly integrated Load Balancer that was capable of operating in the agent society, and capable of delivering near-optimal plans for agent/server assignment using a variety of user-selected strategies.

In our base year, we entered a project that had already been underway for one year, so we had to fit into a project that was already proceeding on a schedule, and we had to fit into that schedule with new interactions and data requirements.

1.1 Queuing Theory vs. Load Balancing Tradeoffs

Our role on the project began with a pre-documentation phase, where we described our existing solver and related technology, and discussed some problems in performance management. The Boeing team had a major task the first year in studying the existing system, talking to people on the team, and writing email messages. I sent out many long and detailed messages describing the types of problems that need to be addressed in designing distributed systems, or agent systems, and I also gave an overview of some of the underlying mathematics involved in solving these problems.

This series of messages on distributed agent system design was then turned into a document which was sent to our DARPA Ultralog Program Manager Mark Greaves and our Contracting Officer Representative Marty Stytz and passed to members of the group for discussion, complete with talking points. A polished version of this lengthy document on techniques in distributed design is available in the appendix, Section 13. This real-time document preparation via the lists followed by a white paper document was useful in describing the current techniques to the Ultralog Scalability Group. It also helped us understand the types of problems and data capabilities that were present in the existing Ultralog architecture, and helped us meet members of the team.

At our first quarterly Ultralog meeting in Albuquerque NM we held extensive discussions about the topics I raised on the email lists regarding performance management and queuing theory. It became obvious that the types of information we would need for a queuing theory model would not be ready by the 2002 testing, and may not be ready by the end of the project. Boeing was not directly dealing with the data acquisition problem, and had to develop tooling for the current agent system using only available data, or minor additions to existing data. A queuing theory model requires a statistical description of job/task arrivals at a resource, including descriptions of the task arrival rate at the processors, and the size of the tasks arriving at the task processors. This task breakdown in terms of arrival rate and size distribution is a fundamental requirement of distributed system design when using queuing theory to design a system (the industry standard). This was somewhat unfortunate, since queuing theory is the "gold standard" in distributed system performance management, and our existing Boeing solver was based on queuing theory. It also meant that we would need to make to make compromises and changes in our plans for performance management.

In our proposal our solver was described as a tool for solving general problems in distributed system design, with a focus on the intractable (NP-Complete) problems in combinatorial optimization in faulty queuing networks. This was our basic building block, and it allowed multi-threaded tracking of sequences of tasks performed in this type of faulty queuing network. The tool was designed to explore the distributed system design space, and hence used queuing theory to estimate wait times in sequences of tasks that were spawned by user-initiated events (start of job, or query). These task sequences could be the threads generated in a typical user generated task, or job, that itself has sub-tasks that are spread around a network, so the overall job or task is solved in a multi-threaded distributed fashion. Our solver would trace through the list of initial tasks (jobs), and find the longest running thread in the set of all threads generated from the initial events that are being tracked for performance and risk. Our original Windows-based solver would then move the subtasks around the modeled network. It would do this for all subtasks in all of the modeled threads, and find a state (subtask/server assignment) which minimized the objective function. This objective function in our proposal was a function of the response times and state risk of the set of modeled tasks.

Since the data items needed to support a full queuing theory model of the Ultralog agent system would not be ready by 2002, this meant that we had a new task direction after the Albuquerque meetings. In our Albuquerque design meetings it was decided that we would pursue a solver that used purely real-time data for a system, and we would not model performance threads. This meant we would have to drop the queuing theory model, and switch to a health model of performance based on mean CPU, Memory, and Bandwidth usages and limits. This also meant that while our projections for state management would likely produce improved performance, it would not be possible to mathematically guarantee a reduction in overall run time, since only a detailed threaded model like our previous model could do that, and the data to feed this type of detailed queuing model would not be available. We do believe, however, that a future system could provide the types of sample thread monitors to make a viable queuing network model, which we will discuss in the later section “Implications for Further Research”.

Once the decision was made to proceed with a real-time load balancing solution versus a more involved queuing theory model, then our project tasks were now fairly well defined for the rest of the project. For the duration of 2002, we would modify our existing Windows-based solver so that it would deal with real-time agent CPU burn rates and memory requirements, where the memory was modeled on a private non-shared basis. On each server on the system, each agent was allowed to specify its current CPU/Memory resource requirement on its current server, and an estimate of its potential CPU/Memory requirement on all other servers in the system for which the agent is eligible for deployment. The inter-agent communications was initially proposed as bytes per second for all communicating agent pairs, but later refined to messages per second (discussed later). While our current Boeing solver is capable of dealing with complex routings in multiply connected networks, our goals in Albuquerque were further refined so that we were to deliver a tool that was to model a single enclave of agents, and not a full society. This enclave of agents would be operating in a switched Ethernet topology, so our general routing and network topology components of our Boeing solver would not be used in building our Load Balancer for Ultralog.

Using this available data set, our new task goals were to build a solver in 2002 that would demonstrate our technology, although loosely integrated with our solver running in the Windows environment on a network communicating with the rest of Ultralog. In 2002 our main task objective was to minimize the risk of the current running state of the Ultralog system. In this case, risk was defined as the overall state probability of failure. This was a good initial goal for our first year, since state risk is directly related to survivability. In a switched Ethernet topology, any time a component (server, link) fails then agents have to move, and agent moves are costly.

In our second year of Ultralog, 2003, our task objectives were now to incorporate the tools we developed as Windows prototypes in 2002 into fully running Ultralog code, and continue developing refinements to our algorithms. During 2003 we focused primarily on building a tightly integrated version of our 2002 Load Balancer, and added a new strategy that minimized a weighted version of agent risk of the system. Our third and final year of our Ultralog contract, 2004, was used to implement true CPU and memory based load balancing across the system and fully test our resulting Load Balancer for final deployment.

2 Technical Problems

After the Albuquerque meeting in 2002, it was apparent that we would need to perform a major architectural rework of our existing solver, since it was based on a multi-threaded queuing network model, and our Ultralog deliverable would not use queuing theory or thread modeling (see Implications for Further Research). We believe that our ability to deliver a high quality working solver after this acknowledged architectural rework serves as a testimonial to the robustness of our solver architecture. While simulated annealing is at times known for being a bit on the slow side, it is quite famous for being robust, and able to withstand major changes in the definition of the problem and still work well and provide useful solutions to mathematically intractable problems (NP-Complete). Since our solver architecture prior to Ultralog was based on a network of communicating objects that moved around, then after Albuquerque 2002 we still had a model of agents and networked resources so the changes were reasonable. By removing the threaded queuing network component from the solver we were able to make significant improvements in speed and reductions in the size of the Load Balancer.

Once we had a better understanding of the customer problem and data capabilities, we still had some major technical problems that needed to be addressed. These fall into three broad categories. In the first category of problems we had the underlying problems in mathematics that were at the foundation of risk and performance management of distributed systems. These problems in computational complexity have historically been the most difficult. With advances in processing speeds, the technology is just now at the point where a commodity processor can solve difficult NP-Complete problems in a reasonable amount of time. As mentioned elsewhere in this document, most of the hard math problems in distributed design and management are either NP-Complete or #P-Complete. This meant we had new real-time constraints and objectives in the agent system that must be included into the solver. In the second category of problems we have the problem of real-time data acquisition. Any type of Load Balancing algorithm, even distributed algorithms, will need data to support the operations. The data acquisition area was not a Boeing task since it had to be provided by the Ultralog architecture, although it was on the critical path for Boeing's Load Balancer development. The third problem category was the sizing of the model. The Load Balancing solver needed to be small enough to run on a small PC.

By its nature, the class of NP-Complete problems are difficult, with solution algorithms that are a bit slow. The class #P-Complete is in some sense even worse. In the class NP-Complete, although it may be hard to find a solution, once a solution has been found it is possible to easily check that the solution is viable - perhaps not optimal, but at least viable and satisfies all of the major constraints. This checking for viability is modeled on a theoretical computer model known as a Deterministic Turing Machine. The problems in the NP-Complete class may be hard, but if a solution candidate is given, it can be verified in a polynomial number of steps on a Deterministic Turing Machine. With problems in the class #P-Complete, where the network reliability problem is found, the solutions themselves are complex, and it is the current majority "belief" in mathematics (unproven) that the solutions in #P-Complete cannot even be checked for validity on a Deterministic Turing Machine in a polynomial number of steps. This has major implications for scaling of algorithms in problems dealing with probabilities, since probabilistic models must enumerate over all states to calculate the true probability of success (i.e., sum the pathsets in a reliability model).

Another significant problem that was addressed in 2004 was the memory size of the solver. This memory problem was manifest in two distinct areas. The problem of inter-agent communications is an N-squared type of problem, where N is the number of agents in the enclave that is modeled by our solver. Since each agent is capable of sending one-way communications to (N - 1) other agents, then there are N * (N-1) one-way communications paths that need to be modeled. BBN was able to alleviate this N-squared scaling problem in data reporting in 2003 by going to a sparse matrix approach for the data mining for the inter-agent messaging data needed by our solver. Our solver needs the inter-agent messaging data to support the "Minimize Remote Messaging Strategy" used by our solver. In this strategy the solver assigns agents to servers in a way that minimizes the amount of overall messaging between servers. Since server messaging is slow, this strategy would reduce delay times in calculations that require significant messaging by confining agents with high communications to the same servers. The inter-agent messaging matrix is vital data in supporting this strategy. The "Minimize Remote Messaging" principle is a classic technique that is commonly used in distributed systems design. It would be effective in managing agent state in those systems where messaging is a major component of the problem compared to lengthy CPU burn times.

We were also able to make significant improvements in memory size of the solver in 2004, which was becoming a problem as the number of agents in an enclave increased. Prior to our efforts at reducing the solver's memory requirement, we would hear frequent reports of memory problems. There were several areas in the code that were contributing to the memory scaling problem, and other areas in the algorithms that contributed to memory, and the previously discussed problems with data mining for inter-agent messaging.

In the actual code for the Load Balancer, there was legacy code and data structures built into the solver that was taking up a considerable amount of space. Recall in the section labeled "Queuing Theory vs. Load Balancing Tradeoffs", that we discussed our architectural conversion from a solver that was designed as a tool for combinatorial optimization in a faulty queuing network, to a tool that was based on real-time resource use by agents. After this conversion there was some remaining legacy code associated with modeling threads (sequences of tasks distributed around the net) that have subtasks assigned to nodes in a network. This logic was intertwined with other parts of the solver, was difficult to remove, and used considerable memory. By 2004 our enclave size (number of agents and servers) had grown to a point where the threaded architecture was a significant factor in memory use, and was removed along with other cleanup efforts in the final year.

Another significant factor in memory use was our "Solver Self-Attack Algorithm". This algorithm will be discussed later in the section titled "Implications for Further Research", since it dealt with the reliability and survivability aspects of the system. The idea here was novel, where the solver both designed optimal states of the system, and designed optimal attacks on the system. It would use this information and play attack games back and forth inside the solver, and search for solutions that were both Robust and that exhibited reduced disruption to the system during the healing process after damage. This meant that we needed a fast real-time estimate of the "frequency of occurrence" of any given link or node in the set of all possible states or failover plans.

In a network this “object frequency of occurrence” problem is related to the well known source-target network reliability problem, where we ask for the probability that a message can be sent from a source to a target, after considering all possible routings in a faulty network. The source-target reliability problem is known to be #P-Complete (not to be confused with NP-Complete). Although the true scalability of both NP-Complete and #P-Complete problems is currently unknown, it is a common belief that these problem types are both intractable, which roughly means that they are believed to scale exponentially in difficulty with respect to the size of the problem (e.g., number of nodes in a network, etc.). Even if the NP-Complete problems are found (unlikely???) someday to have nice fast polynomial solutions, it is still the belief (unproven) that the #P-Complete problems will be intractable, since they appear in some sense to have worse scaling properties than the NP-Complete problems. This is an important and subtle point, and relates to the memory sizing problem of our Load Balancer. These intractable problems (NP-Complete, #P-Complete) are famous in design and performance-risk management of distributed systems, and we were able to overcome them as expected in the NP-Complete case for large societies.

The single state optimization problems addressed by our solver are NP-Complete. In this problem we try to find a state that meets all of the constraints on CPU, Memory, and Bandwidth, and then optimize based on some measure of risk or performance. The optimized state, or output from the solver, is a single state consisting of a set of assignments of agents to servers on the network. The constraint parts of the problem are clear examples of a multi-dimensional bin-packing problem. In a situation where resources are abundant, this is a fairly easy problem even though it is NP-Hard, and there are a variety of fast heuristic techniques that provide good solutions. When resources are marginal, or barely enough to support the overall computing and communications needs, then the full NP-Completeness of the problem is felt, and we need to use special techniques in optimization to find a reasonable solution. *Since Ultralog is designed for hostile war environment, then we must be prepared for situations where resources are abundant, and situations where resources are tight - and in war, an abundant resource situation can quickly turn into a tight resource situation.*

Our solver self-attack algorithm deals with problems that are both NP-Complete and #P-Complete when it generates state recommendations based on reliability and system failover. The problem in scalability of our solver was that our fused annealing of the #P-Complete reliability problem scaled worse than our algorithm to generate a single optimal state. This is to be expected, of course, given the difference between NP-Complete and #P-Complete. In the #P-Complete case, we are required to investigate thousands of cases out of all 10**39 cases (example number, scales like Traveling Salesperson Problem), where “histograms” are kept of the number of times each major physical object (router, link, server, hub...) is used in the set of all possible states of the system. In comparison, the problem of generating a single optimal state is NP-Complete, and delivers a single state as an answer (agent/server assignment). The #P-Complete reliability problem delivers an answer that consists of many states (usually called pathsets) as an answer, so each answer is itself fairly complex. To generate a fast real-time estimate of the frequency of use for each link in the system, we sum the object use for each successful state used by the annealing solver.

Annealing solvers operate by progressively searching a space based on random and downhill moves using a “temperature” parameter. At high temperatures the moves are purely random, and as the temperature is lowered, the downhill moves (improving objective function) are accepted with a probability given by exp(-dE/kT), where dE is the change in value of the objective function (uphill/downhill) after a move, and T is the temperature, and k is a constant. As the temperature is lowered, the moves become progressively downhill or greedy, and theoretically converge to the optimal solution “in distribution” or statistically at low temperature. To get a fast estimate of object frequency of use, we were integrating the annealing algorithm by summing over all states sampled by the algorithm, but weighted in such a fashion that as the temperature is lowered, the contributions are exponentially damped. This exponential damping was used to counterbalance the exponential sampling of high quality states as the temperature is lowered. The result is a quality weighted view of the frequency of use of objects in the solution space, and is given below. In a loose sense, it is probabilistic view of the “good parts” of the solution space.

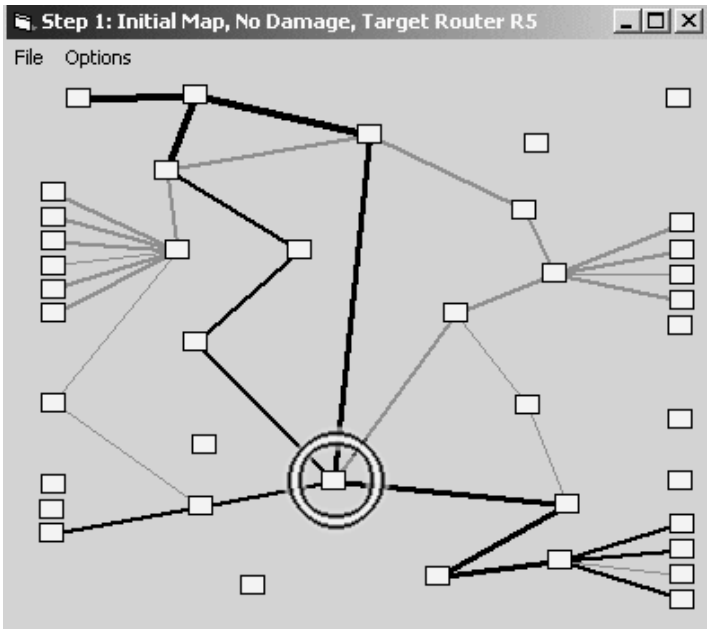


Fig. 1: This figure is a weighted view of the risk-performance solution space, where the links are weighted by counting quality-weighted states. Thick lines/links are used by many high quality solution states, whereas the thin lines are used by only a few states. The dark black lines represent links used by the current optimal state of the system, while the gray lines represent backup and failover possibilities for the system. The circle is highlighting a point in the solver self-attack algorithm.

This quality weighted view could be used by both an attacker and by a designer. A designer (or state optimizing algorithm) would want to design the states using the “thick lines” as they selected objects to support the state. The attacker would also want to attack these same thick lines. In principle, this could be done with different node sizes too, but it gets messy. The thick lines represent high quality solutions that are used in a large number of plans and failover plans. By using objects with thicker lines, the state is less likely to be disrupted by changes elsewhere in the net, since a “thick link” represents many possible states. These thick lines make good targets too for an attacker. Since the thick lines represent many high quality state possibilities, an attack on this link, if used for an operational state, would destroy both the state and a large number of high quality failover states. Our prototype solver self-attack algorithm used this logic iteratively to analyze the solution space, and make recommendations for optimal states based on considerations of how a smart (near optimal) attacker would attack the solution space of performance.

While this attack algorithm appears to have considerable promise for improved survivability, and is discussed in the section on “Implications for Further Research”, it was a major source of the memory and speed problems we were having with the solver. Our primary goal was to generate a fast small solver that address the single state optimization problem with respect to both risk and performance, so we removed our attack algorithm (a stretch goal), and focused on the size and speed of the current solver.

After removing the last threading parts of the code, doing some code cleanup, and removing the solver self-attack algorithm, we were able to get memory reductions on the order of a factor of ten or more over the old code. This was highly significant since it would now allow our Load Balancing solver to run on very small computers, perhaps even palmtops. The memory reduction, speedup, and general code cleanup was the final task of 2004, and we delivered a faster solver that had a variety of strategies, and was a fraction of the original size.

One of the biggest technical problems throughout the project was the ability to get the data needed to support the Load Balancing solver operations. This is a critical problem, since *any* solver, regardless of algorithm, will require data to manage performance. In our first year we discovered that we would not be able to get the data needed to support a standard queuing theory model of performance, so we used real-time performance data. In our second year we were able to integrate our solver into the Ultralog society with real-time CPU and Memory data. In our third year, we had problems getting the inter-agent communications data needed for the “Minimize Remote Messaging Strategy”.

Our Minimize Remote Messaging Strategy was based on minimizing the amount of time spent on inter-host messaging, since messages sent over a LAN or WAN are slow compared to messaging between agents on the same server. This is an effective strategy when there are a lot of small tasks that need to be performed that require significant communications between the tasks. For this strategy to be effective, the inter-agent communications matrix must be based in Bytes/Sec, since heavy traffic requires longer delays. It became apparent in 2003 that we would not be able to get inter-agent traffic in units of Bytes/Sec, but would have to deal with traffic in units of Messages/Sec. This was less desirable for performance management since some messages are larger than others. By 2004 we were starting to get inter-agent traffic from the group that was doing the data mining in units of Messages/Sec, but the data was still a bit dirty. The data at last report was still needing to be cleaned up, but once it is the Load Balancer should be ready to generate optimal agent/server assignments that minimize remote messaging. This feature has been tested on a stand alone basis without the data feed from Ultralog and the Load Balancer is working perfectly. The data feed architecture itself is working well, so the only problem is cleaning up the actual data. Therefore, once the inter-agent messaging data is ready, the Load Balancer will be ready to deliver solutions.

3 General Methodology

Boeing had developed a working software tool for designing distributed systems prior to joining the Ultralog team. Part of the Boeing proposal included porting some of the algorithms and code from the existing Windows-based solver to the Ultralog agent environment. This was a major conversion, since the threading, queuing, and complex routing parts of the solver had to be extracted, and the core solver modified to account for real-time rates of agent CPU and memory usage and inter-agent messaging information. We also had to deploy this new solver capability for use on an as-needed basis as the Load Balancer in the working Ultralog agent environment.

In the first year we continued to develop all of our main code and algorithms in the Windows environment, while modifying the solver for use with real-time agent resource requirements. The communications was accomplished in the first year by providing a web based interface to our Windows solver, where we were able to retrieve input information on agent CPU and memory use from one of the enclave managers in the Ultralog society, and respond after about 5 minutes or less (our target response time) with an optimal assignment of agents to servers. This response was sent back via http to the enclave manager, which would then take this information and deploy agents as specified. This demonstrated our solver, using what is now called our “Minimize State Failure” strategy, but still had our state solver running on a Windows box with an imbedded http server.

In our second and third years of the project, we focused on building our Windows based algorithms into fully integrated Ultralog code written in Java. This was a challenge, since all of our code was written in Visual Basic, and required conversion of the many real time routines that occur in our Windows solver with full GUI and state representation graphics. During this conversion we would be developing new algorithms, strategies, and techniques for our solver in the Windows version of the solver, and porting them to Ultralog during development. This was actually an effective strategy, since it let the person doing the algorithm development focus on that activity, and let the person doing the porting and conversion to Ultralog focus on that effort. As a whole, this strategy was even more effective than we had anticipated before the start of the project.

3.1 Load Balancing Solver

Once we had clear task objectives it was then necessary to convert our solver to the Ultralog environment. The problem was now a clearly defined mathematical problem for the Load Balancer, and we proceeded to develop a number of strategies for our solver. Each strategy had a different objective function, and is documented in the appendices. These objectives dealt with various measures of system performance with strategies like “Load Balancing of CPU and Memory”, and “Minimize Remote Messaging”, and measures of system risk with strategies like “Minimize State Failure” and “Load Balancing of Risk”. The solver also allowed mixed strategies, such as the mixed strategy of “Minimize State Failure + Load Balancing of CPU”.

The mathematical problems that are addressed are problems in constrained optimization similar to a multi-dimensional bin packing, with the constraints operating in Bandwidth, CPU, and Memory, and with the optimization being performed with a number of different objective functions that are available in the various solver strategies.

The Load Balancer uses a Simulated Annealing algorithm which is now a standard technique in global optimization. The strength of the annealing algorithm is that it is extremely robust, and can readily deal with changes in the objective function without having to perform a total rewrite of the solver. When properly designed and tuned, an annealing algorithm can be comparable in speed to other general algorithms in combinatorial optimization. This has been proven in the general case in the now famous “No Free Lunch Theorem”, which deals with the performance of general algorithms, and is discussed in the appendix where we included our first year white paper on distributed system design, Section 13.

The primary design point in any annealing solver is choosing an effective set of state transition moves to search about the space. In the case of Ultralog, the annealing state transitions consisted of transfers of agents from one server to another server, or in some cases exchanging two agents on two different servers. In the solver we would make these types of state transitions in both random and greedy steps. The random steps are obvious, and are needed to keep the global annealing solver from getting stuck in a local minimum. The greedy state transitions were moves where the solver would pick a server, and pick moves for agents on that server that would improve the objective function in a single step. For example, in the greedy state moves for minimizing state risk, we have occasional moves that in one step move all of the agents off of a high risk server and spread them around to the remaining servers in the enclave. Moves like this help the algorithm converge quickly to the desired optimal solution.

4 Technical Results

The effort of turning our distributed system design tool into the agent Load Balancer for managing performance and risk was considerable, but allowed us to produce a solver for state management that is extremely robust and allows a great deal of flexibility for dealing with a variety of situations. The various strategies that are described in our documentation allow the operations of the system to be optimized in a variety of threat and performance environments.

Our solver now appears to perform in a 500+ agent environment, and responds with quality solutions in less than a minute, which is much better than our targeted 5 minute response time. A rough memory estimate shows that a 1000 agent enclave with 200 nodes will require about 39 meg of RAM for our solver. In today’s world of cheap RAM that is tiny. By comparison our solver prior to our memory reduction efforts would have required 1.8 Gig of RAM. Our memory reduction efforts were clearly significant, and should allow future users to run very large societies on small processors. We have not tested a 1000 agent problem with 200 nodes, but we do expect that it would take us back to the five minute response time region for good answers on a 1-2 GHZ computer.

As processing times improve in the next couple of years this type of solver technique should be able to address society enclaves with thousands of agents. If full networking was added, it is foreseeable that a single solver could address the problems in managing state in a multi-enclave society. Our current solver in Windows had complex network topology capabilities, and this would be a good candidate for future work. It is discussed in the section titled “Implications for Further Research”.

Overall, the solver capability should scale like that of solvers for the Traveling Salesperson Problem (TSP). The underlying bin-packing/optimization problems and TSP problems are similar, although more effort is involved in evaluating a state of a distributed system than is involved in calculating the distances in the TSP. We bring up the TSP problem since it is the most famous NP-Complete problem and most technical readers can identify with it. As our processing capability improves we will see both improvements in solving the TSP problem, and improvements in solving large problems in distributed agent design, since they are in the same complexity class.

All of the solver strategies were tested and worked well on a stand-alone basis. The Load Balancer is now fully integrated into the Ultralog society. Some of the data that is fed to the Load Balancer still needs to be cleaned up. This data mining and data cleanup task is being performed by BBN. Once the data is ready for the solver, then the Load Balancer will be ready to perform on all strategies. At this point, the primary strategy with data mining problems is the “Minimize Remote Messaging Strategy”. All other strategies have clean data and are working well.

The overall technical conclusion is that our global solver technique is quite effective for large agent enclaves, and that this situation will only improve with time and processing speeds. As discussed above, the NP-Complete problems associated with single state management and the NP-Hard optimization were well handled by our simulated annealing solver. Furthermore, there are techniques inside the solver that could be extracted for use in distributed algorithms (with some polish) in a future distributed Load Balancer or State Manager. The solver is written in a modular fashion, so it should be fairly easy to add new strategies, objectives, and capabilities to the code.

The Load Balancing solver that was delivered should function without major problems well into the future for large societies. It has also provided a path into the future for further development based on some of the ideas discussed in the section labeled “Implications for Further Research”.

The topics in reliability and the #P-Complete complexity class were partially addressed in our documentation and prototype coding, using our solver self-attack algorithm. This algorithm appears to have tremendous potential for generating battle-hardened systems. However, it took up quite a bit of system memory, and was a bit slow. We believe that this attack algorithm would make an extremely valuable addition to a future Ultralog-like system that is designed for attack.

We have provided a Load Balancer test document in Section 10 that is used to evaluate the stand-alone solver. As mentioned in the section on computational complexity, it is difficult to find near optimal solutions to NP-Complete problems, and this complicates testing. While a solution can be readily checked to ensure that it is valid (meets hard constraints on CPU, memory, bandwidth), it is extremely difficult to test whether a given solution is “optimal”, since you need to know the optimal solution to test a solution candidate.

5 Important Findings and Conclusions

The management of distributed systems in general, including distributed agent systems, involves a number of intractable problems in resource management that must be addressed in near real time. While the designer of a distributed system must confront these intractable problems during the design process, the system designer also has the luxury of being able to study the projected system performance “off-line”. By way of contrast, the problems in managing risk and performance in a distributed system are in many cases the same problems confronted by the system designer, except that these intractable problems must now be addressed in real time to effectively manage the system. This puts a considerable burden on the state optimizer since these design problems are computationally hard, and must be solved within minutes at most.

This demand for a real-time solution has implications on the types of solvers that can be considered for state management. The choice of state management software can be broken down into two broad classes, distributed solvers and centralized global solvers. The debate between global solvers that manage risk and performance by region or group and distributed solvers has been ongoing for years. While distributed solutions scale well, in most cases they are not capable of addressing the core problems in scheduling and mathematics that are at the heart of most problems in distributed design. The main criticism about global techniques, such as those used by our current solver, is that they do not scale well. This is indeed true. The current options are between using distributed techniques that do not address the core hard problems, and global techniques that do address the problem, but that do not scale well to very large problems.

We chose the global techniques since they are now capable of addressing the core hard problems in mathematics for the largest society that we were able to generate in the Ultralog lab. The fact that our global solver can address large problems is highly significant. This situation, where global solvers manage large systems, will continue to improve as the speed of processors improves. While we do expect significant advances in distributed algorithms for performance and risk management, we also believe that the debate between global algorithms and distributed algorithms will enter a new phase as fast processors are developed that are capable of solving extremely large problems in global optimization, and able to manage large distributed systems from one or a few management nodes.

We believe that these results were highly significant, and validated the global solver concept for managing risk and performance of Ultralog. Furthermore, this situation will continue to improve as computer processing speeds increase.

6 Significant Development

During our work on Ultralog we developed working algorithms that were then coded into the Ultralog system. We produced a number of planning and architectural documents during the project. We have included the final Load Balancing sections of the Architectural document for the Robustness Defense Thread (our working group) in the appendix, Section 9 of this current document. The Load Balancer section written by Boeing is actually Section 8 in the Adaptive Robustness Architectural document. The primary products developed by the Boeing team are listed below:

- Load Balancer Software
    - Solver that Manages Risk and Performance for Agents in an Enclave
    - Architecture to include Load Balancer into Ultralog System
- Documentation
    - Reports, Docushare (Ultralog web), Email Archives
    - Paper to HPC 2004
    - Final Architecture document
    - Load Balancer Testing Document

Our primary deliverable is the actual Load Balancer, which is a global optimization routine that takes as input the CPU, memory, bandwidth, link/host failure probabilities, and connectivity of the enclave, and as output provides a set of agent/server assignments. This has been provided as a fully operational tool that can be called at any time, and used in a variety of run scenarios called strategies.

We have listed the “Architecture to include Load Balancer...” as a separate item in the above list under software based on its importance. When we arrived on the project there was no ability to easily acquire real-time data on resource use for CPU, memory, and bandwidth. While a working Load Balancer is important, it has been developed and wired into the Ultralog society in such a way that parts of it can be replaced on a modular basis. We have left in place at the end of Ultralog both a working Load Balancer with multiple strategies, and an architecture for continued development and replacement of the Load Balancer capabilities. Given the importance of state management in performance and risk management, we believe that the current Load Balancer architecture will serve well as Ultralog is developed.

Our other main product is the documentation that supports both the current Load Balancer and Load Balancing architecture. This includes both the final code, algorithm, and architecture documentation, as well as the ongoing project documentation that may be useful in understanding how the team reached some of its design decisions during the project. The Adaptive Robustness Architecture document serves as the user's manual and design document for the Load Balancer. In the architecture document we describe the solver design and the parameters that are involved in interacting with the Load Balancer. For completeness, we have included the Boeing contribution to the Adaptive Robustness Architecture document in the appendix, Section 9. The documentation in the appendix Section 9 is the same as that provided in Section 8 of the Adaptive Robustness Architecture document. The Adaptive Robustness Architecture document is included in its entirety on a CD ROM that includes other documentation deliverables.

Also included in the documentation set is the test document for the solver. Since our solver is primarily an optimization tool, the testing of the solver can only be done in a statistical sense. The clearest test of solver output is for the tester to try to find a solution to the underlying optimization problem that has a better value of the objective function than the one returned by the solver. Since the underlying problems in performance and risk management are NP-Complete, then this is a difficult task. In this case, it is as hard for the tester to find a good solution as it is for the solver to find a good solution. In most cases it will be difficult for a "manual tester" of the system to devise a solution (agent/server assignment) that is better than that delivered by the solver, assuming the same objective function is used by the tester and by the automated solver. Our testing document is presented in the appendix, Section 10, and describes the testing done on the stand-alone solver. In most cases, we perform tests and verify that the reported solution is near the optimal end of the range that is searched by the solver. This has always been the case in the test cases studied.

7 Special Comments

Boeing under spent its contract in the final year by approximately $138,000, which was approximately 30% of its 2004 funding. Since our initial estimate was based on queuing theory and complex networking, then our final solver was simpler than that originally specified in the proposal. This made completion of the Load Balancing task possible at a reduced budget.

In 2003 we contacted Marty Stytz and Mark Greaves to see if we could expand our effort to recapture some of the complexity that was left out after the queuing theory model was set aside. We wrote a proposal for new tasks and funding, but we were unable to expand the project. Given this situation, we proceeded with our reduced model without queuing theory and delivered the final product on time and under budget. In this case we were left with a budget that wasn't quite big enough to take on major new tasks.

Effective state management is the difference between a system operating, and a system operating well. It is the difference between risky operation of a distributed agent system, where servers frequently fail and require costly agent moves, and safe operations where probabilistic failures are minimized. It can also be the difference between a system that is slow and ineffective, and one that is capable of high throughput that meets quality of service constraints. Given the importance of state management, we hope to see increased effort in this area as Ultralog continues into deployment.

Overall I believe that the Load Balancing effort should have been much larger. I was in some sense the only mathematician on the team, or at least the only one that ever talked about math. I wrote Marty Stytz in 2003 to suggest that the importance and difficult of state management (Load Balancing) versus the effort devoted to the task was out of proportion. I believe that effective state management is vital, and the industry is just now getting to the point where global solvers, and effective distributed solvers with hints from global solvers, could manage very large systems. I would like to get DARPA interested in this same idea, and implement some of the improvements discussed in the document and in the next section “Implications for Further Research”.

8 Implications for Further Research

During our time on the Ultralog program we had the opportunity to consider a number of enhancements to the system that were not implemented due to time and resource constraints. In this section we will discuss some of these enhancements, with a view towards future research and development.

Our overall conclusion is that Ultralog as a technology has tremendous potential that far exceeds its current level of funding, both in general for a future Ultralog program, and in particular for the funding needed in the area of State Management. This was our area (Load Balancing), an area that is notoriously complex, and that has tremendous direct impact on the performance and risk of the system. It is our recommendation that Ultralog be extended into a second (perhaps renamed) effort called Ultralog II or likewise. In this project we should build on our existing efforts and capabilities by adding the following features to a Load Balancing effort, which we recommend be renamed to State Management:

- Reliability / Battle Hardened Software
    - Implement our solver self-attack algorithm
    - Attack harden our current techniques using algorithms similar to our solver self-attack algorithm.
- Complex Topological Networks, Complex Routing
    - Improved Failover and routing planning
- Distributed Algorithms
    - Local Heuristics, with distributed decision making
- Queuing Theory
    - Queuing networks used to estimate wait times in distributed threaded tasks
- Model Critical Threads (Agent Task Sequences)
    - Sample Threads feel out performance space, and used in state model
        - Threads selected along communications flows in agent communications diagrams
        - Compare to DARPA Control Plane where selected points/links represent complex network
- Mean System Rehydration Time
    - Model and optimize time spent moving and rehydrating agents
- Distributed Persistence
    - Store agent persistence data at multiple places on net
        - Eliminate single point failure of persistence engine

Battle-Hardened Systems: Solver Self-Attack Algorithm

The critical thing to point out about our solver self-attack algorithm is that it was progressing in the direction of building a battle hardened system. By building on a performance optimization engine, it collected information about the “good parts of the solution space” while it was looking for optimal solutions. The information it collected was based on the frequency of occurrence of a link in the set of all quality weighted solutions. This is valuable information, since it tells an attacker about how often a link is used in the set of failover plans. If you destroy a systems failover capability, you destroy its ability to self-heal, so our solver contained in its workings information on “failover density” for all objects.

As mentioned earlier we can get a fast estimate of this for real time use by integrating our annealing solver as it cools. We use these first estimates of our link weights or failover densities to attack our system, as described in our paper that was given to the High Performance Computing conference in Cetraro Italy in 2004. This paper is given in the appendix, Section 11. By attacking the system at the point (link, server, router) with the highest failover density, we are attacking the system at its most vulnerable point for healing. If you kill a systems ability to heal, you have killed the system.

We do optimal plans for states and optimal attacks against states iteratively in the algorithm, and take the system that is left after an optimal attack and generate a new optimal state of the system. The optimal attack on the state is planned at the same time as the search for the optimal state, since the optimal attack is based on frequency of occurrence of quality weighted solutions in the annealing search. As mentioned previously, an exponential damping is applied during summation over the states in the annealing search, which ensures that the converged solutions at the end of the search do not overwhelm the summation of states found at earlier points in the search before phase transition. As we see by the exponential nature of the annealing state transition probabilities that scale like exp(-dE/kT), we have an exponential sampling of high quality solutions, so our partial exponential damping during the search helps make the overall result reflect the quality solutions seen by the solver as it explores the space.

Using an algorithm like this is designed to produce a “Battle-Hardened System” that is designed to withstand smart attackers. The attack generated by this algorithm generates attack sequences that destroy the current optimal states at the link or node that impacts the largest number of quality failover states. In principle the attack threads can be parallelized for performance. This would allow system designs that meet performance and quality of service demands and yet have been battle-hardened, and designed to withstand smart attacks with minimal impact. This is a radical change from the current Ultralog system. The current system will withstand an attack, but the performance may suffer, or an attack may force major agent moves that would not be necessary if the system were designed to withstand smart attacks. Agent moves are costly, and planning a system state to minimize major disruptions after partial failure could be a useful addition to the next generation Ultralog.

Switched Ethernet Topology vs. Complex Topological Networks

Our current Ultralog Load Balancer is capable of modeling agent performance in a switched Ethernet topology. This is a simple star topology, and is used to model a single enclave of agents. We would like to recommend that the Load Balancer be extended to complex networks. This would allow multiple enclave interactions and complex routing capabilities. Our internal Boeing solver allows modeling of complex networks, and we believe that this capability should be added to Ultralog. This capability would allow true system wide load balancing and performance management across multiple enclaves. This is a major task, but is so central to performance management of large systems that we believe that an extended effort should be made in this area.

A Load Balancer that modeled complex networks could be used for routing management, and enhanced to provide prioritization schemes for quality of service management. It would also be useful to establish some type of hierarchical control of the system, and allow for agents to be paused to allow for partial degradation.

Distributed Algorithms

Distributed algorithms always hold the potential for scalability and speed improvement, although they are at times difficult to design for specific problems. Some local heuristics like exchanging agents between neighboring servers if one of the servers is running hotter are effective. Many of these types of techniques, some of them fairly complex, are used as single state moves in our annealing solver.

Since the solver already has some of this logic in its state transition part of the code, then this would be a good source for local exchange operators in a distributed algorithm that perform moves of agents between a host and its neighbors. This could be implemented in Ultralog by taking any of the annealing operators for a given strategy, and implementing them locally. This is a greedy technique, and in general will not do as well as the global solver technique that is currently used in the Load Balancer, but it does have the advantage of being distributed. Of course there would need to be some “tweaking” of the algorithm and other protocols and event timers to make it work, but this would be effective in building a system. A more involved approach might be a combination of the current global solver used to generate a starting solution, or restart the system at a good healthy point, and then use a distributed Load Balancer or distributed State Manager to let the system slowly evolve on a local distributed level.

An even more polished approach to distributed management might involve using our "performance map" (Fig. 1). This map is a probabilistic view of the frequency of use of high quality solution states. It is a set of numbers between zero and one for each node and link in the system. This map can be generated by our solver as described in the solver self-attack algorithm, and then used locally at each node to make distributed decisions. The performance map could be spread to each node in the system, similar to a pheromone, and used locally along with other heuristics in a distributed algorithm. Since the performance map is generated by our solver self-attack algorithm which attacks the performance solution space, then this would allow our enhanced solver to add "global hints" in the form of the performance map to each node. This would allow each node to make truly local decisions in a distributed fashion. The difference is that this distributed algorithm would take occasional hints from a performance map generated by the global solver. The global solver wouldn't make all of the decisions, but could provide an initial direction in the form of a starting solution, and a hints file that could be propagated to other nodes for later use as the system evolves. This hints file provides the information needed for battle-hardened solutions, since it is built by our attack algorithm and contains valuable information on frequency of link and node use in the space of all solutions. This is vital after attack, since a state that is well positioned according to the weights in the performance map will be less likely to have major disruptions and agent moves after an attack. This global hints file could also be generated on an individual agent basis, although this increase the size of the pheromone by a factor equal to the number of agents being tracked with hints. A global hints file from our attack solver combined with a heuristically based distributed solution may be an effective next generation for the Ultralog Load Balancer.

Queuing Theory

We have already talked about the role of Queuing theory in performance management, and the Boeing solver's use of queuing theory in its performance models of distributed systems. For the Ultralog agent system we did not have access to the information needed for queuing theory. This was mainly a time and resource constraint, since it is possible to extract this type of information for at least some of the agent tasks and interactions. There was a study of an Ultralog-like system by one of the Universities that used queuing networks, so it is feasible, although it is data intensive and complex. We believe that a queuing model might be effective for some of the critical threads of tasks performed in Ultralog. Queuing models are effective in estimating wait times due to random task arrivals and task durations. In a performance model of a large agent system, the task arrivals and durations may appear random, so a queuing model would be reasonable.

Thread Modeling

While it is probably not feasible to model the entire Ultralog system (or any large distributed system) as a multi threaded queuing network, it should be possible to identify critical threads in the overall process and optimize these threads. When there are only a few users or user groups in a distributed system doing a few things, it is easy for the system modeler to generate a simulation model with the anticipated workloads. When a system gets larger we will have to resort to sampling a subset of threads, and using these threads to represent the system. In some sense this is similar to or could be connected to the Control Plane topic at DARPA. In this area we use a set of points in a larger net as control points in a larger net. For a thread modeling task, we would recommend some type of pattern recognition to extract threads of interest for systems modeling.

In 1998 the author of this report gave a talk at an INFORMS conference in Seattle that dealt with the problem of designing Load Tests for large scale distributed systems. An effective Load Test must model a large number of objects in a distributed system. The test of the system should be a good match to the behavior we see in production. In the talk it was proven that the Load Test Design Problem was NP-Complete, and provided a solution technique using simulated annealing. The system was modeled in a 72 dimensional space of Oracle tuning parameters and classic system performance parameters like response time for queries, host CPU utilization, and other metrics. We applied this technique within Boeing to the design of Load Test suites for large scale distributed systems. We were able to reduce the design time from months required to generate low quality tests, to minutes to generate high quality tests that carefully matched the targets and goals for the test. The thread modeling task is a similar problem, where instead of finding a set of jobs to run on a single computer, we need to find a set of job-task sequences, or threads that represent critical sub-operations in an overall program like Ultralog.

One way to do this would be to take the existing agent communications diagrams, and look for patterns along diagrams that show excessive CPU or messaging. This pattern recognition could be automated to extract performance threads of interest. In modern day performance engineering, a computer specialist will create a model of a system and extract well known threads. We are suggesting that quite a bit of this could be automated with pattern recognition using inputs from the existing agent communications diagrams, and the monitored CPU and messaging along these threads. Once a set of performance threads is available, it could be used with or without queuing theory to make a model of performance.

Mean System Rehydration Time.

In the white paper that is provided in the appendix Section 13, I discuss the issue of the time spent moving and rehydrating agents. This would be an excellent addition to a future State Manager. We discussed this at our Ultralog design meetings, and agreed that it would be useful, but that the data would not be available during our time on the Ultralog project. This is easy to understand since BBN was already involved in significant data mining efforts to support our Load Balancer (BBN did a fantastic job). I would like to suggest that in the next Ultralog-like system, that the State Manager (Load Balancer) be assigned the task of assigning agents to servers in such a way that the overall mean time spent moving and rehydrating agents is minimized. This would require additional efforts at data mining to support this task.

Some agents are fairly easy to move, whereas other agents have considerable data needed to store their persistence information. This topic of move and rehydration is discussed in the white paper in Section 13 and would be a useful addition to a future state manager. It also ties in with the next topic, which is distributed persistence.

Distributed Persistence

In 2004 the topic of Distributed Persistence came up after our design meetings. Prof. Subrahmanian of the Univ. of Maryland was working in the area of distributed persistence. For most of the Ultralog project we were to consider the existing persistence engine as “out of bounds” for testing the system for single point sources of failure. The operating system had some backup capability, but our current persistence engine was a single point source of failure. In 2004 we began discussions with Prof. Subrahmanian at the request of Dr. Mark Greaves to investigate this topic. After discussions with Prof. Subrahmanian, we reported to Dr. Greaves that distributed persistence would have been an excellent addition to the Ultralog Load Balancer, but adding something radically new to Ultralog in mid 2004 would be dangerous, and that modifications to the Ultralog persistence engine would be radical if we were to address the topic of distributed persistence.

The topic of distributed persistence and the techniques for moving an agent system from its existing state to a new proposed state are major areas of investigation that warrant a separate project and continued development of Ultralog. The topic of distributed persistence should be stressed since it has a direct impact on survivability.

Future of State Management

We believe that the new topics discussed in this section, when added to the existing Ultralog architecture and state management capabilities, would make a significant advance in the state of the art of distributed computing. A battle hardened system with thread modeling in a faulty queuing network to ensure performance and quality of service would be a valuable addition. It could be enhanced to operate in a distributed mode with hints or total direction from a global solver than accounts for agent rehydration time in its plans for an optimal state. This new state manager would be the difference between a system that survives after an attack but requires major reconfiguration and delays, and a system that is designed to withstand attacks with reduced interruptions, agent movements, and delays. With the addition of a distributed persistence engine, this new system would be truly fault tolerant and battle hardened.

9 Appendices: Load Balancer Portion of Robustness Architecture Document (extract)

This section has been extracted from the full Architecture document that was published by the Adaptive Robustness Defense Thread. In the full Adaptive Robustness document this section is listed as section 8.

Ex Nihilo Global Solver (Load Balancer)

The global solver we are using is designed to optimize standard performance and failure measures of a distributed system by optimally assigning computer jobs or agents to servers.

In 2002 we ported a set of algorithms for distributed system design (used by our ex nihilo tool) to the distributed agent problem. The first version of the agent reliability and performance model was designed to improve survivability by minimizing state failure (which requires agent rehydration), and attempted to improve system response time (like Oplan generation) by keeping all hosts and links in their healthy operating ranges (Max Percent Utilization of CPU, Memory).

In 2004 we develop new strategies designed to improve response time of the system. In the new Minimize Remote Messaging strategy, we are focused on minimizing remote messaging situations, where one group of agents is in frequent communications with another group of agents on another server. In the new Load Balancing of CPU and Memory strategy, we are focused on "true load balancing", where we attempt to use resources in an equal fashion across the system. While the reduction in remote messaging and load leveling is a reasonable design goal for a system, it does not mean that the system will have improved response time. Similarly, our attempts to keep the hosts and links within maximum utilization limits does not mean that the system will have reduced response time. However, by keeping the system operating in a healthy operating range, we frequently do see improvements in response time of a system. We continue to think that our best opportunity for overall system response time improvement is to model a threaded set of parallel calls, and optimize the bottleneck threads, as discussed last year, and as implemented in our solver used outside of Darpa.

In Ultralog, the model is intended to produce near optimal designs in near real time by assigning agents to hosts to minimize a user-selected combination of weighted performance measures that make up the objective function. It should be clear that any model of response time or failure would require a careful analysis of system components, so the inputs to ex nihilo require detailed knowledge of hardware (hosts, data links, topology, failure probabilities), and resource requirements of system components (Total CPU burn rate per agent, inter-agent messaging, etc.).

9.1 Solver Objective Function and Strategies

There are a number of unique metrics like state failure, CPU/Mem load balancing, and agent risk that one may use in measuring the "quality" of a state (agent/host assignment) produced by the load balancer. In this section we describe the load balancer strategies and the associated objective functions for each strategy. As a matter of convention, we are minimizing the objective function, so "downhill moves" in our solver are in the direction of improved quality.

9.1.1 Minimize State Failure Strategy

This strategy will minimize state failure due to component failures anywhere in the system. Each component in the system (nodes, links, routers) is assigned a failure rate (like the familiar 99.99% uptime advertised by web hosts, which means a 0.01% failure rate). To find the state failure rate, we start by finding the probability of the state being "fully operational" in its current mode. This is obtained by multiplying one minus the probability of component failure for all components that are actually used in the state. This quantity is then subtracted from one to find the probability of state failure, which is the quantity being minimized. The goal is to find an agent assignment that removes high risk nodes and links while still satisfying the NP-Complete constraint satisfaction problem. This is a fairly basic definition of state failure, and assumes statistical independence of the nodes and links. Clearly this could be generalized, but it does a good job of capturing the major components of risk in a system. It is important to understand that any failure, anywhere in the state, or multiple failures, are regarded as causing a state failure, with no regard to degree of failure in the model when Minimize State Failure is the sole component of the objective function.

The objective function for this strategy is given as follows (in variables that should be self explanatory):

Obj = Pfail, where Pfail = 1 - Psuccess, and Psuccess is given by

Psuccess = (1-Pfail(node1))*(1-Pfail(nodes2)) *... * (1-Pfail(link1))*(1-Pfail(link2))... * (1-Pfail(router))

The product is over all links, nodes, and routers used in the current mode of system operation. Since the load balancer for Ultralog only treats a single enclave in a switched ethernet topology, then only a single router is included in the above product. The Pfail that is used in the objective function is obviously single mode failure, as opposed to "reliability". Reliability is the number we get when we consider ALL possible modes or states of operation of the system.

9.1.1.1 Using the Minimize State Failure Strategy

In this strategy, we are maximizing the "uptime" of the system in the assigned state. A set of agent server assignments defines (among other things) our state. For each component (link/server/router...) in the state, we take that component's probability of success and multiply it into the overall product of probabilities from other components. The probability of failure is one minus this overall product of Psuccess's, which is the probability of system failure due to any cause. This means that when we minimize failure, we (naturally) try to avoid faulty components. This strategy will eliminate as many faulty hosts and links as possible. A motivation for this type of strategy is that it will tend to minimize agent movement that is forced onto the system by component failure. When components fail (server, link, router), then agents usually have to move somewhere, which is computationally expensive.

9.1.2 Load Balancing of Risk Strategy

This strategy minimizes the degree of "agents at risk" on nodes in the system. For each node in a given state of the system, we begin by calculating the quantity (Number of agents on node) * (Pfail of node). We then minimize the maximum value of this quantity over all nodes in the state. This min/max procedure can be thought of as minimizing the maximum expected damage from a single point of attack on the system, and is again subject to the NP-Complete constraint satisfaction part of the problem. The basic idea here is to "put lots of agents on low risk nodes, and fewer agents on high risk nodes".

The objective function component is given as follows:

Obj = Max(all inod's in system) [Pfail(inod) * NumAgents(inod)]

9.1.2.1 Using the Load Balancing of Risk Strategy

This strategy is designed to minimize the maximum expected damage from a single point of attack, and is a measure of survivability. As a secondary goal in this strategy, we do backfill operations like finding hosts with low agents at risk and add more agents. The strategy is primarily intended to minimize agents at risk, and secondarily to take advantage of unused resources. This would be a strategy to use when there is a higher threat condition, in combination with the "Minimize State Risk" strategy.

9.1.3 Mixed Strategy: Min State Failure + Load Balancing of Risk

This mixed strategy is performed in two passes. In the first pass, the solver minimizes expected state failure and eliminates high risk nodes from the state, as described in the Minimize State Failure strategy. In the second pass, the solver "load levels risk" among the remaining nodes selected in the first pass.

9.1.3.1 Using the Mixed Min State Failure + Load Balancing of Risk Strategy:

This is a fairly natural mixed strategy, which is best used in high threat situations. It will primarily minimize state failure, and secondarily minimize agents at risk.

9.1.4 Minimize Remote Messaging Strategy

This strategy is a classic technique used in distributed system design to improve performance, and is based on graph partitioning. For each agent pair in the system, the model calculates the contribution to inter-host messaging (either bytes/sec or messages/sec). For agents on the same host, this quantity is of course zero. For agent pairs on different hosts, the model adds this contribution to the grand total of inter-host messaging. The solver then attempts to minimize the overall inter-host messaging by choosing a set of optimal agent/server assignments. The goal here is to find an agent assignment that keeps groups of agents with high rates of inter-agent communications on the same physical host while still satisfying the NP-Complete constraint satisfaction problem.

The objective function component is given as follows:

obj = sum (all i != j; inod != jnod) {AgentMsgRate(Agent(i->inod), Agent(j->jnod))}

The notation is used here as follows:

Agent(i->inod) = agent number "i", which has been assigned to node inod. Basically this is just a sum over all agent pairs with agents assigned to different hosts.

9.1.4.1 Using the Minimize Remote Messaging Strategy

This strategy attempts to assign agents to servers in a way that minimizes overall interhost messaging. A frequent source of performance problems is interhost messaging, so this strategy may be useful when messaging delays are longer than expected computational tasks (e.g., many small programs talking to each other over a slow WAN).

9.1.5 Mixed Strategy: Min State Failure + Min Remote Messaging

This mixed strategy is performed in two passes. In the first pass, the solver minimizes expected state failure and eliminates high risk nodes from the state, as described in the Minimize State Failure strategy. In the second pass, the solver minimizes remote messaging among the remaining nodes selected in the first pass.

9.1.5.1 Using the Mixed Min State Failure + Min Remote Messaging Strategy

This mixed strategy is designed to primarily minimize state failure, and secondarily improve performance by reducing remote messaging.

9.1.6 Hamming Metric Strategy

This strategy is used to find solutions that are "close" to some other target solution. The target solution is usually understood to be that of a previous operating state of the system. The Hamming distance between any given state and some other state is the number of places where the two states disagree on agent/server assignment. For example the Hamming distance is two if a given state differs from some target state in two places, where two agents are assigned to different servers.

The objective function component is given as follows:

Obj = sum (i) {Fdiff(i)}, where

Fdiff(i) = 1 if inod from the assigned state i->inod is not equal to inodH from the hamming target state, iHamming->inodH. Otherwise, Fdiff(i) = 0

9.1.6.1 Using the Hamming Metric Strategy

This strategy could be useful in minimizing needless moves after a host failure as a short term goal in getting the system operational, followed by later adjustments by other Load Balancer strategies like "Minimize Remote Messaging" or “Load Balancing of CPU and Memory” that are focused on longer term targets for the system. By minimizing agent moves, we are attempting to reduce the move and rehydration times associated with a state change.

9.1.7 Mixed Strategy: Min State Failure + Hamming Metric Messaging

This mixed strategy is performed in two passes. In the first pass, the solver minimizes expected state failure and eliminates high risk nodes from the system, as described in the Minimize State Failure strategy. In the second pass, the solver finds the solution closest to the Hamming target state, subject to the node set selected in the first pass.

9.1.7.1 Using the Mixed Strategy: Min State Failure + Hamming Metric Messaging

This mixed strategy is primarily focused on reducing the state failure rate, and secondarily in minimizing the number of agent moves required to migrate to the minimum failure state.

9.1.8 CPU / Memory Load Balancing Strategy

This strategy is designed to minimize load imbalances in CPU and memory for the hosts involved in supporting the system. This is clearly a problem in multiobjective optimization, with the solver attempting to minimize the combination of CPU and memory in a two dimensional space. The objective function is based on minimizing the sum of the weighted utilization differences in CPU and memory for all host pairs in the system, and is given below:

Obj = sum (i,j, i!=j) {sqr(A*(Ucpu(i)-Ucpu(j))**2 + (1-A)*(Umem(i)-Umem(j))**2)}

where,

Ucpu(i) = CPU utilization of host i

Umem(i) = Memory utilization of host i

A = weighting coefficient between 0 and 1

9.1.8.1 Using the CPU / Memory Load Balancing Strategy

This strategy might be chosen if the overall system risk is low, in an attempt to improve performance.

9.1.9 Mixed Strategy: CPU/Mem Balancing + Min Remote Messaging

This mixed strategy is performed in two passes. In the first pass, the solver performs a load balance across the set of eligible servers. During this first pass, the solver tracks the range of balance sampled during the pass. In a second pass, the solver performs a Min Remote Messaging strategy, while attempting to keep the overall Load Balance in the optimal part of its range detected in the first pass. This is accomplished by adding a penalty factor to the second pass remote messaging objective function. There is no penalty when the load balance in the second pass is within 10% of the best found load balance of the first pass. Above this 10% level, there is an exponential penalty factor that is equal to one (no penalty) at the 10% level, and ramps up to a factor of exp(2) at the maximum (least optimal) value of the first pass load balancing range. This forces the min remote messaging strategy to both optimize remote messaging, but stay with the best 10% of the load balancing range, thus mixing the two strategies.

9.1.9.1 Using the Mixed Strategy: CPU/Mem Balancing + Min Remote Messaging

This mixed strategy is focused on performance. It emphasizes the load balancing strategy, and uses the Min Remote Messaging strategy as a refinement. It should be used when the primary area of interest is performance of the system, with risk not being a major consideration.

9.2 Agent/Function DEFINITION:

The ex nihilo tool was designed to model systems of special user-defined functions, and how they call each other and place loads on a distributed system. A function may burn CPU on a host, or call another set of subroutines or "subfunctions", at specified calling rates.

For purposes of the Ultralog application, a function is a simple performance model of an agent that uses system resources. A function call would correspond to an agent posting a request on a blackboard, which is acted upon by another agent or plugin. The actual functions may be aggregations of sets of agents, as discussed below in the "modeling tips" section.

For any given function ***F***, we will need a list of "subfunctions" that are called by ***F***. Each function may call multiple subfunctions. The main properties of these "user-defined functions" are listed below

9.2.1 Functions in Objective Function Job Set:

Set of functions being traced, required input

The user of ex nihilo must select a set of agents or jobs/functions to be included in the "objective function" used by the optimizer. We will refer to this set of user-selected jobs as the ***Objective Function Job Set***. It is the set of jobs that generate the objective function. Each job/function in the Objective Function Job Set is evaluated in terms of Probability of Failure, and modeled for messaging and host CPU and Memory constraint satisfaction. Functions not selected as Active members of the Objective Function Job Set are ignored.

9.2.2 Agent Host Properties:

Agents in ex nihilo are modeled as processes on hosts with a specified rate of resource consumption. To fully define an agent requires a list of all hosts on which the agent may run, along with CPU, memory, and messaging,

The ex nihilo tool models agent memory on a private, non-shared basis.

9.2.2.1 Agent Host Eligibility

(0,1) flag to indicate availability on hosts, required input

For each agent ***F***, we need the set of hosts on which the agent ***F*** may run. Note that eligibility does not imply that the solver will actually choose to execute the function/agent on a given eligible server. A function Fb called FROM a server Sa will be assigned to execute on a server Sb based on the current state matrix. The agent host eligibility matrix is the primary vehicle for cross thread interactions for Security. If an agent is forbidden to reside on a specific host due to Security constraints, then the agent/host eligibility should be set to zero, otherwise it should be set to one.

9.2.2.2 Agent CPU Consumption Rate on Current Host

Agent % CPU Utilization, in units of (CPU Sec) / (Real Sec), required input

For each agent in the system, we need the CPU consumption rate by that agent on the host on which it is currently executing. This CPU Consumption Rate is the agents percent utilization of CPU on the current server, and should be specified in CPU Seconds per Second. If Agent CPU Consumption Rate on Current Host is provided in terms of JIPS (Java Instructions per Second), then a conversion factor is required for the conversion from JIPS to CPU Seconds per Second for each host on the system.

9.2.2.3 Agent CPU Consumption Rate on Alternate Hosts

Agent % CPU Utilization, (CPU Sec) / (Real Sec), required input

For each agent in the system, we need an estimate of the CPU consumption by that agent on ALL hosts for which that agent is eligible for execution. This is needed to compare the CPU consumption rate on the current host to the CPU consumption rate on alternate hosts. Note that in a homogeneous server environment (like a lab test with all servers equal), the Agent CPU consumption rate will be the same on all hosts.

9.2.2.4 Conversion Factor from JIPS to CPU Seconds per Second

JIPS per CPU Sec/Sec, optional/required input

This is required when the agent CPU consumption is specified in units of JIPS. The conversion factor must be given for all Hosts in the modeled system. The JIPS to CPU conversion is handled internally by the interface between the Ultralog enclave manager and the Load Balancer.

9.2.2.5 Agent Private Memory Requirements

MB Memory per Agent, required input

For each agent, the tool requires an estimate of the amount of private memory, in bytes, used by that agent. The memory usage for an agent should be determined by the level of memory usage before onset of paging. This memory allocation should be considered private to that agent, and not shared with other agents on the same host. The problem of deciding whether a collection of agents with varying memory requirements can be distributed among a set of servers with memory limits is a version of the famous "bin packing problem", and is known to be NP-Complete. If the answer to the bin packing problem is NO (no distribution of agents with all bin limits satisfied), then part of the distributed system will be forced into using paged memory, which can adversely affect performance.

9.2.3 Agent Communications Properties

Communications between agents is modeled as the current rate of messaging between all host pairs. When an agent moves from one host to another host, this move will potentially impact any and all network links required to maintain communications at the specified rate.

9.2.3.1 Agent-to-Agent Communications Rate

Bytes/Sec, or Messages/Sec, required input

For each agent pair in the system, we need the rate of communications, in KiloBits/Sec or Messages/Sec, between the two agents. If possible, when the expected message transit times are long compared to the message initiation and setup time, then this input variable should be in units of KB/Sec which may be more tightly correlated with performance than Messages/Sec. If the time to prepare an interhost message is longer than the transmission time, then we might want to use Messages/Sec as input.

9.3 Host Properties:

A host in an ex nihilo design is actually just a connection point in a network, and may be one of four broad categories of hardware devices (Server, Shared LAN Hub, Router, Virtual Link Node).

9.3.1.1 Host Type:

"Router" or "Server", required input

For the Ultralog Load Balancer, we are only concerned with Routers and Servers. Note that a switched Ethernet can be modeled with a router object and directed Link objects.

9.3.1.2 Host Name:

ASCII Text String, required input

The ***Host Name*** property is the text name of the host object. For example, a server associated with processing paychecks might be labeled "servername-paycheck". Any ASCII characters, including spaces, are allowed in the Name property.

9.3.1.3 Host Number:

Integer Agent ID, required input

Each host in the network has a unique positive integer identifier, referred to as the ***Host Number***. If there are "numhost" hosts in the current design, then the number must be between 1 and numhost. All IO, including flat file input and output is listed in order of Host Number. The Host Number is generated internally by the solver for each host of an agent enclave, so is not a strictly required input.

9.3.1.4 Host Message Forwarding:

{0,1} flag for {False, True}, required input

The ***Message Forwarding*** property determines whether the host is allowed to forward IP packets in a network environment. If the value of message forwarding is set equal to "1", then IP forwarding is allowed. If the value is set equal to "0", then IP forwarding is not allowed. Always set packet forwarding to "1" for Routers and LAN hubs. For servers, set the value to "0".

9.3.1.5 Host Failure Probability:

[0,1] probability, required input

The ***Host Failure Probability*** property is the probability of failure of the host, in some (unspecified) user measurable time T. For example, 0.001 is a common failure rate for commercially advertised hosting services (non critical service).

9.3.1.6 Host Maximum CPU Utilization:

Percent of Total CPU Capacity, required input

For each host in the system, we need to know the maximum CPU utilization limit for that host. This limit is typically set in the range of 80-90% utilization by most sysops. Beyond the 90% utilization levels, we frequently experience long waiting times in queues.

9.3.1.7 Host Background CPU Utilization:

Percent of Total CPU Capacity, required input

For each host in the system, we need to know the background load on that host, in units of percent of overall host CPU capacity.

9.3.1.8 Host Memory:

MB Memory Capacity of Host, required input

For each host, we need to know the amount of physical memory present in that host, in units of Megabytes (MB).

9.4 Data Link Properties:

An ex nihilo Data Link object is used to model communications between host objects. The link is characterized by the starting and ending host numbers of the objects it connects, Link Bandwidth and Background Loads, and the probability of link failure.

9.4.1.1 Link Type:

"Switched Ethernet", required input

All links modeled by the Ultralog Load Balancer will be modeled as links in a switched Ethernet (directed links).

9.4.1.2 Link Name:

ASCII Text String, required input

The ***Link Name*** property is just the text name of the data link object. For example, a link connecting server S1 to server S2 might be named "S1=>S2". Any ASCII characters, including spaces, are allowed in the Name property.

9.4.1.3 Link Starting Node:

Integer Host ID, required input

The Starting Node is the Host Number at the start of a Directed Link. For Shared Hub Links, this must always be the Host Number of the Host connected to the Hub.

9.4.1.4 Link Ending Node:

Integer Host ID, required input

The Ending Node is the Host Number at the end of a Directed Link. For Shared Hub Links, this must always be the Host Number of the Hub.

9.4.1.5 Link Probability of Failure:

[0,1] probability, required input

The ***Link Probability of Failure*** property is the probability of failure of the Data Link, in some (unspecified) user measurable time T.

9.4.1.6 Link Bandwidth:

KBits/Sec, required input

For directional links, the ***Link Bandwidth*** property is the one-way bandwidth of the data link. For Shared Hub links, this is the bandwidth rating of the Hub object (assuming the bandwidth ratings of the cabling and network cards support the hub bandwidth rating).

9.4.1.7 Link Background Load:

Percent of Total Link Capacity, required input

For each link in the system, we need the background load on that link, in units of percent of overall link bandwidth.

9.5 Miscellaneous Run Parameters:

In addition to defining Hosts, Data Links, and Functions used in an ex nihilo run, the model also requires a small set of Miscellaneous Run Parameters to fully specify the problem. These run parameters are used to define the goals and weights in the objective function, and specify initial starting configurations (initial state), times for input/output, solver options, and a few GUI parameters.

9.5.1.1 Initial Starting Solution:

Agent/Host Assignment Array, required input

At the start of the solution process, ex nihilo requires an ***Initial Starting Solution***, or seed solution, to begin the search. For our demonstration, the Initial Starting Solution will be the current state of the system, where agents have been assigned to specified hosts. For the "Static Layout" use case, the initial starting solution will be assumed to be random.

CurrentHost(agentid) = Host id of the Host where agent "agentid" is currently running.

9.5.1.2 Min and Max constraints on Failure:

[0,1] probability, required input

The ***Max*** constraint is the dividing line between acceptable and unacceptable solutions. For component values less than the Max limit, the solution is acceptable. For values greater than the Max limit, the solution is unacceptable. The ***Min*** constraint is the dividing line between the set of "equally good" solutions at or below the lower limit, and solutions which have variations in goodness.

9.5.1.3 Polling Time for Ex Nihilo Inputs:

Number of Seconds

To operate in near-real-time, it will be necessary for ex nihilo to periodically check for updates to key inputs to the model. Any of the key inputs could potentially be redefined during the model run. A balance is needed between minimizing interruptions to the solver (low polling rate), and responding to system changes in a timely manner (high polling rate).

9.5.1.4 Clock Time for Required Ex Nihilo Outputs:

Clock Time When Outputs are Due

In addition to polling for changes to ex nihilo input, the model will require a target time when Ultralog expects an answer from the ex nihilo solver.

9.6 Ex Nihilo Outputs

Although there is a large amount of information available as output from an ex nihilo run, we are predominantly interested in the Agent/Host recommendation, Probability of Systems Failure, total inter-host messaging rates, and Utilization Levels on Systems Components (CPU, Memory, Traffic).

9.6.1.1 Agent/Host Recommendation:

Host-id for each Agent

Each Agent is assigned a unique Host for agent hosting services. This output is in the form of an array, RecommendedHost(iagent), that provides the Host recommendation for agent number iagent.

9.6.1.2 RecommendedHost(iagent) = Host recommendation for agent number iagentNumber of Solver Increments:

Integer

Log File Variable: **SystemDesign.annealForm_,iterations_**

This is the number of steps performed by the annealing solver. For an enclave with about 500 agents, this number should be in the thousands of increments.

9.6.1.3 Probability of State Failure:

Probability, [0, 1]

Log File Variable: **(1 - psuccess)**

This is the overall probability of systems failure, which is one minus the probability of success. The probability of success is defined as the probability that there are no failures in any part of the current state (no failure-induced movement of agents required). The resulting failure is the failure rate for the single operational state of the system, and includes host, link and hub failures, but does not account for all backup modes or failover modes of operation. It is important to distinguish between the *failure rate* of the current operational state, and the *reliability*, which is the probability of success after considering ALL possible failover modes of operation

9.6.1.4 Hamming Distance to Target State:

Integer Hamming Distance, [0, NumAgents]

Log File Variable: **hamming**

This is the Hamming distance between the user-specified target state and the state found by the load balancer. It is the number of agents that differ in agent/host assignment between the target state and the load balancer state. A low value of hamming means that the solver found a state close to the target state.

9.6.1.5 CPU/Memory Load Balance:

Real

Log File Variable: **loadbal**

The “loadbal” variable is the value of the CPU/Memory load balancing component of the objective function. A low value of loadbal represent good balance between CPU and Memory across the set of hosts available for use in the state.

9.6.1.6 Agent Risk:

Integer, [0, NumAgents]

Log File Variable: **pfailuresurvive**

The “pfailuresurvive” variable is the value of agent weighted risk component of the objective function. A low value of pfailuresurvive means that the solver has been successful at placing most agents on low risk servers and few agents on high risk servers.

9.6.1.7 State Failure Rate:

Probability, [0, 1]

Log File Variable: **(1 - psuccess)**

The state failure rate is one minus the probability that all state components are operational (1-psuccess).

9.6.1.8 Remote Traffic:

Real

Log File Variable: **remotetraffic**

The “remotetraffic” variable is the value of all interhost message (either messages or bytes per second), and is a component of the objective functino. A low value of remotetraffic means that agent pairs with heavy communication are colocated, minimizing messaging delays.

9.6.1.9 Hamming Sample Maximum:

Integer Hamming Distance, [0, NumAgents]

Log File Variable: **hammingsamplemax**

This is the maximum value of all Hamming distances between all sampled solution states found by the load balancer and the user-specified target state used for the Hamming test.

9.6.1.10 Hamming Sample Minimum:

Integer Hamming Distance, [0, NumAgents]

Log File Variable: **hammingsamplemin**

This is the minimum value of all Hamming distances between all sampled solution states found by the load balancer and the user-specified target state used for the Hamming test.

9.6.1.11 CPU/Mem Load Balancing Sample Maximum: First Pass of Two Pass Strategy

Real

Log File Variable: **loadbalmaxfirstpass**

This variable is used in two-pass load balancer strategies, and is used to report results from the first pass of a two-pass run. At the end of a two pass run, loadbalmaxfirstpass is the maximum value of the CPU/Memory Load Balancing (loadbal) component of the state objective function found in the first pass.

9.6.1.12 CPU/Mem Load Balancing Sample Minimum: First Pass of Two Pass Strategy

Real

Log File Variable: **loadbalminfirstpass**

This variable is used in two-pass load balancer strategies, and is used to report results from the first pass of a two-pass run. At the end of a two pass run, loadbalminfirstpass is the minimum value of the CPU/Memory Load Balancing (loadbal) component of the state objective function found in the first pass.

9.6.1.13 CPU/Memory Load Balancing Sample Maximum:

Real

Log File Variable: **loadbalsamplemax**

This is the maximum value of the CPU/Mem Load Balancing component of the objective function, where the maximum is taken over all states sampled by the solver.

9.6.1.14 CPU/Memory Load Balancing Sample Minimum:

Real

Log File Variable: **loadbalsamplemin**

This is the minimum value of the CPU/Mem Load Balancing component of the objective function, where the minimum is taken over all states sampled by the solver.

9.6.1.15 State Failure Rate Sample Maximum:

Probability, [0, 1]

Log File Variable: **pfailuresamplemax**

This is the maximum value of the State Failure Rate component of the objective function, where the maximum is taken over all states sampled by the solver.

9.6.1.16 State Failure Rate Sample Minimum:

Probability, [0, 1]

Log File Variable: **pfailuresamplemin**

This is the minimum value of the State Failure Rate component of the objective function, where the minimum is taken over all states sampled by the solver.

9.6.1.17 Agent Risk Sample Maximum:

Integer, [0, NumAgents]

Log File Variable: **pfailuresurvivesamplemax**

This is the maximum value of the Agent Risk component of the objective function, where the maximum is taken over all states sampled by the solver.

9.6.1.18 Agent Risk Sample Minimum:

Integer, [0, NumAgents]

Log File Variable: **pfailuresurvivesamplemin**

This is the minimum value of the Agent Risk component of the objective function, where the minimum is taken over all states sampled by the solver.

9.6.1.19 Remote Traffic Sample Maximum:

Real

Log File Variable: **remotetrafficsamplemax**

This is the maximum value of the Remote Traffic component of the objective function, where the maximum is taken over all states sampled by the solver.

9.6.1.20 Remote Traffic Sample Minimum:

Real

Log File Variable: **remotetrafficsamplemin**

This is the minimum value of the Remote Traffic component of the objective function, where the minimum is taken over all states sampled by the solver.

9.7 Load Balancer Control Flags: Pure Strategies

9.7.1.1 Hamming Strategy Flag:

Boolean (True/False)

Control Variable: **hammingCheck**

If the hammingCheck variable is True the Load Balancer will use the Hamming Strategy in its search, and find a state that minimizes the objective function variable "**hamming**". This will find a state close to the user specified Hamming target state.

9.7.1.2 CPU/Memory Load Balancing Strategy Flag:

Boolean (True/False)

Control Variable: **loadbalCheck**

If the loadbalCheck variable is True the Load Balancer will use the CPU/Memory Load Balancing Strategy in its search, and find a state that minimizes the objective function variable "**loadbal**".

9.7.1.3 Minimize State Failure Strategy Flag:

Boolean (True/False)

Control Variable: **pfailureCheck**

If the pfailureCheck variable is True the Load Balancer will use the Minimize State Failure Strategy in its search, and find a state that minimizes the objective function variable "**pfailure = (1 - psuccess)**".

9.7.1.4 Minimize Agent Risk Strategy Flag:

Boolean (True/False)

Control Variable: **pfailuresurviveCheck**

If the pfailuresurviveCheck variable is True the Load Balancer will use the Minimize Agent Risk Strategy in its search, and find a state that minimizes the objective function variable "**pfailuresurvive**".

9.7.1.5 Minimize Remote Messaging Strategy Flag:

Boolean (True/False)

Control Variable: **remotetrafficCheck**

If the remotetrafficCheck variable is True the Load Balancer will use the Minimize Remote Messaging Strategy in its search, and find a state that minimizes the objective function variable "**remotetraffic**".

9.8 Load Balancer Control Flags: Blended Two-Pass Strategies

9.8.1.1 Blend CPU/Mem Load Balancing + Min Remote Messaging Strategy Flag:

Boolean (True/False)

Control Variable: **blendloadbalmessaging**

If the blendloadbalmessaging variable is True the Load Balancer will perform a two pass strategy. In the first pass the solver will use the CPU/Mem Load Balancing strategy. In the second pass the solver will use a Min Remote Messaging Strategy. This is a mixed performance strategy which emphasizes CPU/Mem load balancing and secondarily minimizes remote messaging.

9.8.1.2 Blend Min State Failure + CPU/Mem Load Balancing Strategy Flag:

Boolean (True/False)

Control Variable: **blendpfailloadbal**

If the blendpfailloadbal variable is True the Load Balancer will perform a two pass strategy. In the first pass the solver will use the Minimize State Failure Rate Strategy. In the second pass the solver will use a CPU/Mem Load Balancing strategy. This is a mixed risk/performance strategy which emphasizes the reduction in state failure (avoid risky servers and links) and secondarily performs the CPU/Mem Load Balancing Strategy.

9.8.1.3 Blend Min State Failure + Min Remote Messaging Strategy Flag:

Boolean (True/False)

Control Variable: **blendpfailmessaging**

If the blendpfailmessaging variable is True the Load Balancer will perform a two pass strategy. In the first pass the solver will use the Minimize State Failure Rate Strategy. In the second pass the solver will use the Min Remote Messaging Strategy. This is a mixed risk/performance strategy which emphasizes the reduction in state failure (avoid risky servers and links) and secondarily performs the Min Remote Messaging Strategy.

9.8.1.4 Blend Min State Failure + Load Balancing of Risk Strategy Flag:

Boolean (True/False)

Control Variable: **blendpfailsurvive**

If the blendpfailsurvive variable is True the Load Balancer will perform a two pass strategy. In the first pass the solver will use the Minimize State Failure Rate Strategy. In the second pass the solver will use the Load Balancing of Risk Strategy. This is a mixed risk strategy which emphasizes the reduction in state failure (avoid risky servers and links) and secondarily performs the Load Balancing of Risk Strategy (Minimize Agent Risk).

9.9 Miscellaneous Parameters

9.9.1.1 CPU/Mem Load Balancing Weight Factor:

Real (0 - 1) (memory - cpu)

Input Variable: **loadbalcpumemratio**

This is the factor for weighting CPU balance versus Memory balance when doing a load balance. A value of 1.0 means that the solver will perform a pure CPU load balance with memory constraints. A value of 0.0 means that the solver will perform a pure memory balance with CPU constraints. Values in between have weighted mixes of CPU and Memory balance.

9.9.1.2 Minimum Allowable Nodes:

Integer (0 - NumNodes)

Input Variable: **MinUltralogNodes**

This is the minimum number of nodes/hosts that will be considered in a search for viable a viable state (agent/host assigntments).

9.10 EN4J: Overview

Ex Nihilo (a.k.a. “EN”) was originally developed in Microsoft Visual Basic (VB), to take advantage of VB’s rapid-prototyping features. This codebase was ported directly into Java, using a few Boeing-built tools. This ensured that the functionality of the two systems was as close as possible to each other, and also allowed relatively simple updating of the Java code, since development of EN in VB was ongoing. The Java version of EN is called EN4J, which stands for “Ex Nihilo for Java”.

The Java code thus produced was re-factored into a set of classes appropriate for use in the Cougaar architecture. In addition, several classes had to be created – ahem – ex nihilo, that is, out of nothing, in order to facilitate interoperation with Cougaar. The end result, a Cougaar “plugin”, along with its supporting classes, constitutes the Java implementation of Boeing’s Ex Nihilo product.

9.11 Conversion of code from Visual Basic

9.11.1 Automated steps

The class **`com.boeing.pw.mct.vb2j`** will process Visual Basic (version 5.0, perhaps others) "form" source files (with extension ".frm") or "Basic" source file (.bas), and output Java source code. It can be invoked via the Java interpreter thusly:

`java com.boeing.pw.mct.vb2j.VB2J -i` *`infile`* **`-o`** *`outfile`*

where

`infile` is the base filename of the form, that is, not including the .frm extension

`outfile` is the full filename to be output, such as "`MyClass_take_1.java`"

(this allows you to do multiple tries at conversion, specifying various output files for the trial results)

The code is converted on a line-by-line basis, in one pass. The converter does not convert non-code resources such as forms and form elements Each line of new Java code is output and includes a "//"-style comment containing the original VB code, to assist in locating and fixing translation errors, and in updating the code.

A summary of the conversion is output to the console and the log file. Here's an example run:

```
java -classpath build com.boeing.pw.mct.vb2j.VB2J -i annealForm -o AnnealForm1.java
VB2J 1.0a; robert.e.cranfill@boeing.com; (c)2003 The Boeing Company
Skipped 1627 lines of header.
First non-header line is: Const qdim = 100
doDo: Wrong number of tokens for 'Do' on line number 1973
doDo: Wrong number of tokens for 'Do' on line number 1997
doDo: Wrong number of tokens for 'Do' on line number 2018
doDo: Wrong number of tokens for 'Do' on line number 2049
doDo: Wrong number of tokens for 'Do' on line number 2069
doDo: Wrong number of tokens for 'Do' on line number 3317
----------------------
Lines processed: 5730
         Fatals: 169
       Warnings: 733
          GoTos: 6
----------------------
```

At the end of the output Java code there is a comment block containing various metrics regarding the conversion, including a list of tokens that were identified as possible external functions:

```
/*
  End translated code for annealForm
     Lines processed: 5730
              Fatals: 169
            Warnings: 733
               GoTos: 6

  Possible functions:
getrandomstate()
getrandomthread()
...
*/
```

These possible external functions need to be looked at on a case-by-case basis. Some will be built-in VB functions, others will be functions in the code being converted.

Any lines that cannot be translated will be output like

```
VBJ2_XLATE_FAILURE;  // {VB2J [1669]} Dim memlistid(numnodemax) As
Long
```

where the token VBJ2_XLATE_FAILURE is purposely designed to make the Java compiler emit an error, so the code must be fixed by hand. The comment indicates the source line number in the VB code, and the text of the un-translated line itself.

Visual Basic tokens that are handled properly are:

DIM, CONST, IF, ELSEIF, END, PRIVATE, SUB, FOR, NEXT, EXIT, ELSE, DO, LOOP and code labels.

GOTO is not translated.

9.11.2 Manual steps

As noted in section 2.1, several things need to be done to produce the final Java code:

- GOTOs must be re-coded, as Java has no support for GOTO. This generally can be done with a 'while' loop and one or more 'break' statements.

- External functions, as noted at the bottom of the converted code, must be identified and called properly

- Any other mistranslations, as noted by the **VBJ2_XLATE_FAILURE** token.

9.11.3 Miscellany

The class **com.boeing.pw.mct.vb2j.VB2JUtils** was initially conceived of as a place for various utility routines for implementing VB functions, but this never panned out, and the class isn't used much, if at all.

9.12 Cougaar-izing the code

9.12.1 Packages

The EN4J code is packaged into various, uh, Java packages, according to its functionality:

com.boeing.pw.mct.vb2j contains the Boeing code for performing VB-to-Java translation. This is not strictly part of the Ultralog/Cougaar deliverable, and is not used in the Cougaar society at runtime.

org.cougaar.robustness.exnihilo contains the objects translated from VB to Java.

org.cougaar.robustness.exnihilo.lbviz contains objects for doing load balancing visualization.

org.cougaar.robustness.exnihilo.plugin contains objects for using EN in a Cougaar environment.

test.org.cougaar.robustness.exnihilo contains JTest classes for automated testing. These are in various degrees of completeness and usability.

9.12.2 Objects

Each VB form and .bas file was translated into a Java object. In addition, VB "TYPE" objects (which are basically just data structures) were also turned into objects of their own, such the "linktype" structure found in "systemdesignx.bas", which was turned into the Java class `Link` (the word "Type" was dropped from all such structures, as it seemed redundant with the idea of a Java class).

In the EN package itself, the main class of note is the SystemDesign class; the .runConfig() method solves the current system as represented in the internal state of the SystemDesign object. In the context of Cougaar, the top-level object is En4JPlugin, which subclasses the appropriate superclass and implements the required Cougaar interface.

A LoadBalanceRequest object was created that encapsulates various parameters required when a data analysis is requested - most importantly, the "annealing time", or amount of time to allow the solver to run before yielding a "best" answer. The LoadBalanceRequest object's presence on the plugin's blackboard is what begins a solver run.

9.12.3 Events

As described in the previous section, a LoadBalanceRequest object's presence on the EN4JPlugin's blackboard is what begins a solver run.

The plugin can emit various Cougaar STATUS events:

"EN4JPlugin: Got load balance request:"
Will show the details of the LoadBalanceRequest object. This should always appear.

"EN4JPlugin: Finished load balance; posting solution:"
"EN4JPlugin: Solution within constraints?" + withinLimits
"EN4JPlugin: .execute exiting normally."
These should always appear, and the 'withinLimits' should be 'true'.

"EN4JPlugin: .execute exiting abnormally; see logfiles."
"EN4JPlugin: Error: Data fails 'checkProject'! See logfile for more info. Exiting."
"EN4JPlugin: Warning: UNSUPPORTED SOLVER MODE:"
An error occured; see the Cougaar log files for full info. It may be necessary to re-run with DEBUG-level logging.

"EN4JPlugin: Warning: Cannot find annealing solution"
This could be an error, or not – it meant the solver couldn't find a solution, and sometimes there is no solution within the given constraints. You'll need to check the log files and look at the configuration to decide.

9.12.4 Logging

The plugin can log copious messages to the log files if asked to, using the standard Cougaar logfile configuration scheme to set the level of messages desired.

9.13 Plugin Configuration

The Cougaar system property

org.cougaar.robustness.exnihilo.plugin.COLLECT_MESSAGE_TRAFFIC

can be set to either ‘true’ or ‘false’. If true, the Cougaar metrics service will be used to continually gather message traffic information, and that information will be used by the solver. If this flag is ‘false’, then the solver will use data provided in the **LoadBalanceRequest** object.

9.14 Testing

9.14.1 Stand-alone Testing

EN4J can be tested by itself, that is, outside of the Cougaar environment. The class **test.org.cougaar.robustness.exnihilo.SystemDesignTest** can be invoked from a command line (that is, in a stand-alone Java VM) and accepts numerous parameters:

```
Java SystemDesign
        {annealSeconds}
        ([{XMLinputfile}] | [-en {inputbasename}])
             [-r]
        [-v]
        [-t {testCaseNumber}]
        [-p]
        [-k {killedList}]
        [-l {logLevel}]

The ‘AnnealSeconds’ parameter and either an XML (the default) or EN-style input
file are mandatory.

The optional flags are:

-en
        Use EN-style input files (.functiondef, .linkdef, and .nodedef files, as
defined and used by the EN VB code)

-r {n}
        Repeat test n times; if r given and no n, then n=100.

-v
        Verify the test case results, if possible (not all test cases can be
automatically verified). Will print out ‘pass’ or ‘fail’ accordingly. See each
test case as described for ‘-t’ parameter.

-t {n}
        testCaseNumber, 1 <= n <= 7. Each test case uses a particular solver mode,
as enumerated in the original EN documentation - but the numbers there and here no
longer correspond. Look in SystemDesignTest itself for a description of each test
case.

-p
        Tests the EN4JPlugin plugin wrapper in addition to the solver itself.

-k {killedNode1[,killedNode2...]}
        Specifies a killed node list, comma separated, each name can be a quoted
string (if it has embedded spaces). Only valid if testing with the Plugin, via –p
option.

-d {xmlFilenameBefore} {xmlFilenameAfter}
        Create a “Load Balance” display using the input XML files given. This uses
the CSVisualizer object and can be used for ‘Viewfoil Engineering’.
```

9.14.2 Testing in a Cougaar Environment

Testing within the Cougaar framework can be done via a servlet, **EN4JTestServlet**, that will generate a **LoadBalanceRequest** object and publish it to the blackboard, but it is perhaps not as full-featured as one might want. It uses an XML file as input, which must be available in the local filesystem of the running society. The format of the XML file is as detailed in the **CSParser** (“Cougaar Society Parser”) object; see the method **testXMLInputStream()** for an example. The result will be displayed via the **CSVisualizer** class, as well as textually (well, HTML-ally).

9.15 EN4J - Appendices

9.15.1 Required Support Packages

The most arcane of the required support packages is the JAR file

```
Scalability_inoether_utilities.jar
```

which was produced by InfoEther in the second year of the program. It provides an object hierarchy that EN uses to model the Cougaar society, and methods for manipulating such.

The standard Cougaar libraries:

```
community.jar
core.jar
servlet.jar
util.jar
```

The “Robustness” Defense Thread’s library:

```
Robustness_bbn_bbn_HEAD_XXX.jar
```
where XXX is the version info

Also required are two Sun-created Java packages:

```
jai_codec.jar
jai_core.jar
```
containing Java Advanced Imaging support for the CSVisualizer.

9.15.2 Abbreviations

Cougaar: Cognitive Agent Architecture, an Open Source project. See http://www.cougaar.org

EN: Ex Nihilo, a Boeing product.

EN4J: The Java implementation of Ex Nihilo for the Ultralog program.

VB: Visual Basic, a Microsoft product

10 Appendices: Load Balancer Test Document

This appendix includes our testing document that was used for stand-alone testing of our Load Balancing solver. It is posted to docushare in the Robustness thread area with a filename of “en2.20-testplan-120504.doc”.

Test Cases For UL Load Balancer:

Summary of Objective Function Components

In this section we describe the load balancer strategies and the associated objective functions for each strategy. As a matter of convention, we are minimizing the objective function, so “downhill moves” in our solver are in the direction of improved quality.

Minimize State Failure

This strategy will minimize state failure due to component failures anywhere in the system. Each component in the system (nodes, links, routers) is assigned a failure rate (like the familiar 99.99% uptime advertised by web hosts, which means a 0.01% failure rate). To find the state failure rate, we start by finding the probability of the state being "fully operational" in its current mode. This is obtained by multiplying one minus the probability of component failure for all components that are actually used in the state. This quantity is then subtracted from one to find the probability of state failure, which is the quantity being minimized. The goal is to find an agent assignment that removes high risk nodes and links while still satisfying the NP-Complete constraint satisfaction problem. This is a fairly basic definition of state failure, and assumes statistical independence of the nodes and links. Clearly this could be generalized, but it does a good job of capturing the major components of risk in a system. It is important to understand that any failure, anywhere in the state, or multiple failures, are regarded as causing a state failure, with no regard to degree of failure in the model when Minimize State Failure is the sole component of the objective function.

The objective function for this strategy is given as follows (in variables that should be self explanatory):

Obj = Pfail, where Pfail = 1 - Psuccess, and Psuccess is given by

Psuccess = (1-Pfail(node1))*(1-Pfail(nodes2)) *... * (1-Pfail(link1))*(1-Pfail(link2))... * (1-Pfail(router))

The product is over all links, nodes, and routers used in the current mode of system operation. Since the load balancer for Ultralog only treats a single enclave in a switched ethernet topology, then only a single router is included in the above product. The Pfail that is used in the objective function is obviously single mode failure, as opposed to "reliability". Reliability is the number we get when we consider ALL possible modes or states of operation of the system.

Using the Minimize State Failure Strategy:

In this strategy, we are maximizing the "uptime" of the system in the assigned state. A set of agent server assignments defines (among other things) our state. For each component (link/server/router...) in the state, we take that component's probability of success and multiply it into the overall product of probabilities from other components. The probability of failure is one minus this overall product of Psuccess's, which is the probability of system failure due to any cause. This means that when we minimize failure, we (naturally) try to avoid faulty components. This strategy will eliminate as many faulty hosts and links as possible. A motivation for this type of strategy is that it will tend to minimize agent movement that is forced onto the system by component failure. When components fail (server, link, router), then agents usually have to move somewhere, which is computationally expensive.

Load Balancing of Risk (Survivability)

This strategy minimizes the degree of "agents at risk" on nodes in the system. For each node in a given state of the system, we begin by calculating the quantity (Number of agents on node) * (Pfail of node). We then minimize the maximum value of this quantity over all nodes in the state. This min/max procedure can be thought of as minimizing the maximum expected damage from a single point of attack on the system, and is again subject to the NP-Complete constraint satisfaction part of the problem. The basic idea here is to "put lots of agents on low risk nodes, and fewer agents on high risk nodes". Note that this concept could be readily generalized (although it has not been) to minimize the maximum expected damage from "multiple points of attack on the system".

This measure (along with Hamming) could also be generalized (not in the schedule) to include measures related to agent rehydration. In general the system should be modeled in terms of "Mean System Rehydration Time" described in previous writeups (once the estimates of individual agent rehydration and move times are available, which is a UL data problem). The system should then be moved into states that minimize the expected time spent in agent move and rehydration.

The objective function component is given as follows:

Obj = Max(all inod's in system) [Pfail(inod) * NumAgents(inod)]

Using the Load Balancing of Risk Strategy:

This strategy is designed to minimize the maximum expected damage from a single point of attack, and is a measure of survivability. As a secondary goal in this strategy, we do backfill operations like finding hosts with low agents at risk and add more agents. The strategy is primarily intended to minimize agents at risk, and secondarily to take advantage of unused resources. This would be a strategy to use when there is a higher threat condition, in combination with the "Minimize State Risk" strategy.

Mixed Strategy: Min State Failure + Load Balancing of Risk

This mixed strategy is performed in two passes. In the first pass, the solver minimizes expected state failure and eliminates high risk nodes from the state, as described in the Minimize State Failure strategy. In the second pass, the solver "load levels risk" among the remaining nodes selected in the first pass.

Using the mixed Minimize State Failure + Load Balancing of Risk Strategy:

This is a fairly natural mixed strategy, which is best used in high threat situations. It will primarily minimize state failure, and secondarily minimize agents at risk.

Minimize Remote Messaging

This strategy is a classic technique used in distributed system design to improve performance, and is based on graph partitioning. For each agent pair in the system, the model calculates the contribution to inter-host messaging. For agents on the same host, this quantity is of course zero. For agent pairs on different hosts, the model adds this contribution to the grand total of inter-host messaging. The solver then attempts to minimize the overall inter-host messaging by choosing a set of optimal agent/server assignments. The goal here is to find an agent assignment that keeps groups of agents with high rates of inter-agent communications on the same physical host while still satisfying the NP-Complete constraint satisfaction problem.

The objective function component is given as follows:

obj = sum (all i != j; inod != jnod) {AgentMsgRate(Agent(i->inod), Agent(j->jnod))}

The notation is used here as follows:

Agent(i->inod) = agent number "i", which has been assigned to node inod. Basically this is just a sum over all agent pairs with agents assigned to different hosts.

Using the Minimize Remote Messaging Strategy:

This strategy attempts to assign agents to servers in a way that minimizes overall interhost messaging. A frequent source of performance problems is interhost messaging, so this strategy may be useful when messaging delays are longer than expected computational tasks (e.g., many small programs talking to each other over a slow WAN).

Mixed Strategy: Min State Failure + Min Remote Messaging

This mixed strategy is performed in two passes. In the first pass, the solver minimizes expected state failure and eliminates high risk nodes from the state, as described in the Minimize State Failure strategy. In the second pass, the solver minimizes remote messaging among the remaining nodes selected in the first pass.

Using the Minimize Remote Messaging Strategy:

This mixed strategy is designed to primarily minimize state failure, and secondarily improve performance by reducing remote messaging.

Soft Constraints

The NP-Complete constraint satisfaction problem at the servers was treated as a hard constraint in 2002. This means that violations of CPU and memory were strictly enforced at all servers. In 2002, if the solver was unable to find a solution that satisfied the hard constraints, then the solver would report a "solution not found" error and terminate. Note that systems with violations in hard CPU constraints should be expected to exhibit poor performance, with excessive queuing and context switching penalties between multiple processes on an overloaded server. Systems with violations of hard memory constraints should be expected to exhibit excessive paging (page faults). The solver has been modified to search for solutions that "minimally violate constraints". This is done in multiple passes through the solver, with each pass attempting to satisfy the least violated constraint.

Hamming Metric Strategy

This strategy is used to find solutions that are "close" to some other target solution. The target solution is usually understood to be that of a previous operating state of the system. The Hamming distance between any given state and some other state is the number of places where the two states disagree on agent/server assignment. For example the Hamming distance is two if a given state differs from some target state in two places, where two agents are assigned to different servers.

The objective function component is given as follows:

Obj = sum (i) {Fdiff(i)}, where

Fdiff(i) = 1 if inod from the assigned state i->inod is not equal to inodH from the hamming target state, iHamming->inodH. Otherwise, Fdiff(i) = 0

Using the Hamming Metric Strategy:

This strategy could be useful in minimizing needless moves after a host failure as a short term goal in getting the system operational, followed by later adjustments by other Load Balancer strategies like "Minimize Remote Messaging" or “Load Balancing of CPU and Memory” that are focused on longer term targets for the system. By minimizing agent moves, we are attempting to reduce the move and rehydration times associated with a state change.

Mixed Strategy: Min State Failure + Hamming Metric Strategy

This mixed strategy is performed in two passes. In the first pass, the solver minimizes expected state failure and eliminates high risk nodes from the system, as described in the Minimize State Failure strategy. In the second pass, the solver finds the solution closest to the Hamming target state, subject to the node set selected in the first pass.

Using the Mixed Minimize State Failure + Hamming Metric Strategy:

This mixed strategy is primarily focused on reducing the state failure rate, and secondarily in minimizing the number of agent moves required to migrate to the minimum failure state.

Load Balancing of CPU and Memory

This strategy is designed to minimize load imbalances in CPU and memory for the hosts involved in supporting the system. This is clearly a problem in multiobjective optimization, with the solver attempting to minimize the combination of CPU and memory in a two dimensional space. While balancing CPU and memory in a system is frequently used in an attempt to improve performance, it must be emphasized that there is frequently little if any relationship between load balancing and response time of a selected set of tasks. The objective function is based on minimizing the sum of the weighted utilization differences in CPU and memory for all host pairs in the system, and is given below:

Obj = sum (i,j, i!=j) {sqr(A*(Ucpu(i)-Ucpu(j))**2 + (1-A)*(Umem(i)-Umem(j))**2)}

where,

Ucpu(i) = CPU utilization of host i

Umem(i) = Memory utilization of host i

A = weighting coefficient between 0 and 1

Using the Load Balancing of CPU and Memory Strategy:

This is the "pure load balancing" that was never part of the solver strategy set. Since Ultralog is focused on Survivability, then our primary load balancing goal in the first two years was health management (CPU, Memory), and risk reduction. The defense coordinator program might choose the Load Balancing of CPU and Memory strategy if the overall system risk is low in an attempt to improve performance.

Mixed Strategy: Load Balancing + Min Remote Messaging

This mixed strategy is performed in two passes. In the first pass, the solver performs a load balance across the set of eligible servers. During this first pass, the solver tracks the range of balance sampled during the pass. In a second pass, the solver performs a Min Remote Messaging strategy, while attempting to keep the overall Load Balance in the optimal part of its range detected in the first pass. This is accomplished by adding a penalty factor to the second pass remote messaging objective function. There is no penalty when the load balance in the second pass is within 10% of the best found load balance of the first pass. Above this 10% level, there is an exponential penalty factor that is equal to one (no penalty) at the 10% level, and ramps up to a factor of exp(2) at the maximum (least optimal) value of the first pass load balancing range. This forces the min remote messaging strategy to both optimize remote messaging, but stay with the best 10% of the load balancing range, thus mixing the two strategies.

Using the Mixed Load Balancing + Min Remote Messaging Strategy:

This mixed strategy is focused on performance. It emphasizes the load balancing strategy, and uses the Min Remote Messaging strategy as a refinement. It should be used when the primary area of interest is performance of the system, with risk not being a consideration.

General Note on Mixed Strategies:

A variety of mixed strategies will be implemented to help deal with problems with conflicting objectives. An example of conflicting objectives would be the case “Minimize Remote Messaging” compared to “Load Balancing of CPU and Memory”. In the Minimize Remote Messaging case, the goal would be to assign agent pairs with high communications rates to the same server, while the “Load Balancing of CPU and Memory” may wish to assign these jobs to separate servers to minimize load imbalances. The Hamming strategy is in clear conflict with other strategies, since any attempt by other strategies to move agents away from their previously assigned host will decrease the quality of the Hamming metric. One of the most reasonable mixes of strategies is the mix of “Minimize System Failure” with other strategies in the list, and will be delivered in the final version of the Load Balancer. We will also attempt to mix other strategies and goals in an attempt to address the conflicting objective problem present with all strategy pairs.

Specific Load Balancer Tests:

In this section we outline several specific tests for the load balancer, and give results from a typical run for each of these tests. In each test, the solver should be able to reproduce the primary objective of the test within a reasonable tolerance, although some of the non-primary objectives may differ considerably. For example, in Test 2, Minimum State Failure, the tests should be able to produce the primary objective function component for the test, which is State Failure Rate given by the variable (1-psuccess), although the tests will not necessarily reproduce the other objective function components, like Load Balance, which is given by the variable loadbal. In each test, the primary objective function component is listed in bold in the "Solution Parameters" section of the test description. For example in Test 2, there is a bold line for "**State Failure Rate"** in the Solution Parameters to signal that this is the primary objective for the test, and should be reproduced by the test. The list of key input and output variables for the Load Balancer are given below:

The key Strategy tests listed below for the Load Balancer are highlighted by listing the test title in *Italics*.

Key Output Variables:

Objective Function Components (LOG FILE):

hamming (0-NumAgents) = Hamming

loadbal = Load Balance

pfailuresurvive (0-NumAgents) = Agent Risk

1 - psuccess (0-1) = State Failure Rate

remotetraffic = Remote Traffic

Objective (objx) = weighted function of above components. This is just a local variable in annealingstepcommand, called objx. The key components of the objective that are of interest to users are listed in bold in this section (e.g., loadbal, remotetraffic, pfailuresurvive, psuccess, hamming). This is not printed to the log file.

Objective Function Sample Ranges (LOG FILE):

hammingsamplemax = Hamming Sample Maximum

hammingsamplemin = Hamming Sample Minimum

loadbalmaxfirstpass = Load Balancing Sample Maximum from First Pass of Two Pass Strategy

loadbalminfirstpass = Load Balancing Sample Minimum from First Pass of Two Pass Strategy

loadbalsamplemax = Load Balancing Sample Maximum

loadbalsamplemin = Load Balancing Sample Minimum

pfailuresamplemax = State Failure Sample Maximum

pfailuresamplemin = State Failure Sample Minimum

pfailuresurvivesamplemax = Agent Risk Sample Maximum

pfailuresurvivesamplemin = Agent Risk Sample Minimum

remotetrafficsamplemax = Remote Messaging Sample Maximum

remotetrafficsamplemin = Remote Messaging Sample Minimum

Key Input Variables (Strategy and Program Control Flags):

Pure Strategy Flags:

hammingCheck (True/False) = Flag for Hamming Strategy

loadbalCheck (True/False) = Flag for Load Balancing of CPU/Memory Strategy

pfailureCheck (True/False) = Flag for Minimum State Failure Strategy

pfailuresurviveCheck (True/False) = Flag for Load Balancing of Risk Strategy

remotetrafficCheck (True/False) = Flag for Min Remote Messaging Strategy

Blended Strategy Flags:

blendloadbalmessaging = Mixed 2 pass strategy flag for "Load Balance CPU/Mem + Messaging" (Blend Load Balance and Messaging)

blendpfailloadbal = Mixed 2 pass strategy flag for "Min State Failure + Load Balance CPU/Mem"

blendpfailmessaging = Mixed 2 pass strategy flag for "Min State Failure + Min Remote Messaging"

blendpfailsurvive = Mixed 2 pass strategy flag for "Min State Failure + Load Balance risk"

Miscellaneous Parameters:

loadbalcpumemratio (0 - 1, memory - cpu) = Load Balancing CPU/Memory weighting factor. (0.0 means pure memory balance with CPU constraints), (1.0 means pure CPU balance with memory constraints)

MinUltralogNodes = Minimum Allowable Nodes

SystemDesign.annealForm_iterations_ = Number of Increments used by solver to find a solution

Test 1: Load Balancing of CPU and Memory Strategy

Subtest 1-1: CPU balance

INPUT:

Set Load Balancing CPU/Memory = 1.0 (pure CPU balance with memory constraints)

Strategy = Load Balancing of CPU and Memory

loadbalCheck = True

loadbalcpumemratio = 1

MinUltralogNodes = 1

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

All Nodes Used, balanced according to cpu, no attempt at memory balance:

OUTPUT FILES: a1.lb-loadbalcpu.state, a1.lb-loadbalcpu.hamming

Server Summary:

Fwd-A	40.0 cpu	23.4 mem	4.0 risk	8 agents
Fwd-B:	40.0 cpu	23.4 mem	6.4 risk	8 agents
Fwd-C	40.0 cpu	46.9 mem	1.6 risk	8 agents
Fwd-D	40.0 cpu	46.9 mem	1.6 risk	8 agents
Fwd-E	40.0 cpu	46.9 mem	1.6 risk	8 agents
Fwd-F	37.5 cpu	87.9 mem	1.5 risk	15 agents

Enclave2 Manager 50.0 cpu 50.0 mem 0.1 risk 1 agent

Solution Parameters:

Load Balance = 0.262 (min=0.262, max=1.858)

Remote Traffic = 6.86 E2

Agent Risk (0-NumAgents)= 6.4

State Failure Rate (0-1) = 0.959

Hamming (0-NumAgents)= NA

Increments = 3,196

Objective = 1.87 E-2

COMMENTS:

It is important to note that the above example (as with all tests) are outputs from an actual run. Since the solver is non-deterministic, and searches for *sub-optimal* solutions, then these results may vary from run to run. For example, in sub-test1-1, the server Fwd-F shows a CPU utilization of 37.5%, with the other servers (except the enclave manager) showing utilizations of 40%. Since it is a non-deterministic search, we cannot rule out that another server might show this low utilization, as opposed to Fwd-F showing the low CPU usage.

Subtest 1-2: Memory balance

INPUT:

Set Load Balancing CPU/Memory = 0.0 (pure memory balance with CPU constraints)

Strategy = Load Balancing of CPU and Memory

loadbalCheck = True

loadbalcpumemratio = 0

MinUltralogNodes = 1

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

All Nodes Used, balanced according to memory, no attempt at cpu balance:

OUTPUT FILES: a1.lb-loadbalmem.state, a1.lb-loadbalmem.hamming

Server Summary:

Fwd-A	65.0 cpu	38.1 mem	6.5 risk 13 agents
Fwd-B	70.0 cpu	41.0 mem	11.2 risk 14 agents
Fwd-C	35.0 cpu	41.0 mem	1.4 risk 7 agents
Fwd-D	35.0 cpu	41.0 mem	1.4 risk 7 agents
Fwd-E	35.0 cpu	41.0 mem	1.4 risk 7 agents
Fwd-F	17.5 cpu	41.0 mem	0.7 risk 7 agents
Enclave2 Manager	50.0 cpu	50.0 mem	0.1 risk 1 agent

Solution Parameters:

Load Balance = 0.243 (min=0.243, max=0.610)

Remote Traffic = 6.40 E2

Agent Risk (0-NumAgents)= 11.2

State Failure Rate (0-1) = 0.958

Hamming (0-NumAgents)= NA

Increments = 3,192

Objective = 1.73 E-2

Subtest 1-3: CPU + Memory balance

INPUT:

Set Load Balancing CPU/Memory = 0.5 (mixed CPU and Memory balance)

Strategy = Load Balancing of CPU and Memory

loadbalCheck = True

loadbalcpumemratio = 0.5

MinUltralogNodes = 1

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

All Nodes Used, mixed balance according to CPU and memory:

OUTPUT FILES: a1.lb-loadbalcpumem.state, a1.lb-loadbalcpumem.hamming

Server Summary:

Fwd-A	55.0% cpu	32.2% mem	5.5 risk	11 agents
Fwd-B	50.0% cpu	29.3% mem	8.0 risk	10 agents
Fwd-C	40.0% cpu	46.9% mem	1.6 risk	8 agents
Fwd-D	40.0% cpu	46.9% mem	1.6 risk	8 agents
Fwd-E	40.0% cpu	46.9% mem	1.6 risk	8 agents
Fwd-F	25.0% cpu	58.6% mem	1.0 risk	10 agents
Enclave2 Manager	50.0% cpu	50.0% mem	0.1 risk	1 agent

Solution Parameters:

Load Balance = 0.654 (min=0.654, max=1.51)

Remote Traffic = 7.22 E2

Agent Risk (0-NumAgents)= 8.0

State Failure Rate (0-1) = 0.959

Hamming (0-NumAgents)= NA

Increments = 3,073

Objective = 4.66 E-2

COMMENTS:

In test 1-3, we have an example of mixed objective optimization. In all problems with mixed objectives, it is reasonable to expect that results might vary due to tradeoffs in the objective, for example, the tradeoff between balancing CPU and balancing memory. In this example, the load balancer has reached a reasonable compromise between balancing memory and balancing CPU.

Test 2: Minimize State Failure Strategy

INPUT:

Strategy = Minimize State Failure

MinUltralogNodes = 1

pfailureCheck = True

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

Avoids FWD-A (pfail=0.5) and FWD-B (pfail=0.8), and then uses all other nodes.

OUTPUT FILES: a1.lb-minfail.state, a1.lb-minfail.hamming

Server Summary:

Fwd-A			
Fwd-B			
Fwd-C	80.0% cpu	93.8% mem	3.2 risk 16 agents
Fwd-D	80.0% cpu	93.8% mem	0.8 risk 8 agents
Fwd-E	70.0% cpu	82.0% mem	2.8 risk 14 agents
Fwd-F	20.0% cpu	46.9% mem	0.8 risk 8 agents
Enclave2 Manager	100% cpu	55.9% mem	0.2 risk 2 agents

Solution Parameters:

Load Balance = 2.62

Remote Traffic = 666

Agent Risk (0-NumAgents)= 3.2

State Failure Rate (0-1) = 0.586 (min 0.586, max 0.959)

Hamming (0-NumAgents)= NA

Increments = 3,386

Objective = 1.17

Test 3: Min State Failure Strategy, MinUltralogNodes Constraint

INPUT:

Strategy = Mininum State Failure

MinUltralogNodes = 6 (There are 7 nodes total in the system)

pfailureCheck = True

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

Avoids FWD-B (pfail=0.8), and then uses all other nodes.

OUTPUT FILES: a1.lb-minfail-minnodes6.state, a1.lb-minfail-minnodes6.hamming

Server Summary:

Fwd-A	30.0%cpu	17.6% mem	3.0 risk	6 agents
Fwd-B				
Fwd-C	65.0% cpu	76.2% mem	2.6 risk	13 agents
Fwd-D	45.0% cpu	52.7% mem	1.8 risk	9 agents
Fwd-E	50.0% cpu	58.6% mem	2.0 risk	10 agents
Fwd-F	40.0% cpu	93.8% mem	1.6 risk	16 agents
Enclave2 Manager	100% cpu	55.9% mem	0.2 risk	2 agents

Solution Parameters:

Load Balance = 2.00

Remote Traffic = 622

Agent Risk (0-NumAgents)= 3.0

State Failure Rate (0-1) = 0.793 (min 0.586, max 0.959)

Hamming (0-NumAgents)= NA

Increments = 3,170

Objective = 1.58

Test 4: Load Balancing of Risk Strategy

INPUT:

Strategy = Load Balance Risk

MinUltralogNodes = 1

pfailuresurviveCheck = True

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

All Nodes used according to min(pfail(inod) * numagents(inod))

OUTPUT FILES: a1.lb-agentrisk.state, a1.lb-agentrisk.hamming

Server Summary:

Fwd-A	20.0% cpu	11.7% mem	2.0 risk 4 agents
Fwd-B	10.0% cpu	5.86% mem	1.6 risk 2 agents
Fwd-C	50.0% cpu	58.6% mem	2.0 risk 10 agents
Fwd-D	55.0% cpu	64.5% mem	2.2 risk 11 agents
Fwd-E	50.0% cpu	58.6% mem	2.0 risk 10 agents
Fwd-F	42.5% cpu	99.6% mem	1.7 risk 17 agents
Enclave2 Manager	100% cpu	55.9% mem	0.2 risk 2 agents

Solution Parameters:

Load Balance = 1.872

Remote Traffic = 692

Agent Risk (0-NumAgents)= 2.2 (2.2 min, 4.0 max)

State Failure Rate (0-1) = 0.959

Hamming (0-NumAgents)= NA

Increments = 3,357

Objective = 7.85E-2

Test 5: Mixed Strategy: System Failure + Load Balanced Risk

INPUT:

Set “Blend Pfail and Survivability” flag in annealing form to flag two pass run

blendpfailsurvive = True

MinUltralogNodes = 1

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

Avoids FWD-A (pfail=0.8) and FWD-B (pfail=0.5), and then uses all other nodes, with “load balanced risk” among the remaining used nodes.

OUTPUT FILES: a1.lb-2pass-minfail-agentrisk.state, a1.lb-2pass-minfail-agentrisk.hamming

Server Summary:

Fwd-A				
Fwd-B				
Fwd-C	65.0% cpu	76.2% mem	2.6 risk	13 agents
Fwd-D	65.0% cpu	76.2% mem	2.6 risk	13 agents
Fwd-E	55.0% cpu	64.5% mem	2.2 risk	11 agents
Fwd-F	42.5% cpu	99.6% mem	1.7 risk	17 agents
Enclave2 Manager	100% cpu	55.9% mem	0.2 risk	2 agents

Solution Parameters:

Load Balance = 0.957

Remote Traffic = 677

Agent Risk (0-NumAgents)= 2.6 final pass (2.6 min final pass, 3.4 max final pass)

State Failure Rate (0-1) = 0.586 final pass (0.586 min first pass, 0.959 max first pass)

Hamming (0-NumAgents)= NA

Increments = 6,825 (two passes)

Objective = 9.28 E-2

Test 6: Minimize Remote Messaging Strategy

INPUT:

MinUltralogNodes = 1

remotetrafficCheck = True

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

Avoids FWD-B (pfail=0.8), and then uses all other nodes, with “Minimum Remote Messaging” among the remaining used nodes.

OUTPUT FILES: a1.lb-minmsg.state, a1.lb-minmsg.hamming

Server Summary:

Fwd-A	95.0% cpu	55.7% mem	9.5 risk	19 agents
Fwd-B				
Fwd-C	85.0% cpu	99.6% mem	3.4 risk	17 agents
Fwd-D	85.0% cpu	99.6% mem	3.4 risk	17 agents
Fwd-E	5.0% cpu	5.86% mem	0.2 risk	1 agent
Fwd-F	2.5% cpu	5.86% mem	0.1 risk	1 agent
Enclave2 Manager	50.0% cpu	50.0% mem	0.1 risk	1 agent

Solution Parameters:

Load Balance = 2.79

Remote Traffic = 179 (179 min, 769 max)

Agent Risk (0-NumAgents)= 9.5

State Failure Rate (0-1) = 0.793

Hamming (0-NumAgents)= NA

Increments = 4,395

Objective = 3.58 E-2

Statistics:

Load Balance: (mean=2.74, sd=0.294, sterr=0.0416, min=2.04, max=3.15, Nsample=50)

Remote Traffic: (mean=159.02, sd=22.732, sterr=3.215, min=126, max=223, Nsample=50)

Agent Risk (0-NumAgents): (mean=8.65, sd=1.905, sterr=0.269, min=3.4, max=9.5, Nsample=50)

State Failure Rate (0-1): (mean=0.799, sd=0.0611, sterr=0.00864, min=0.741, max=0.958, Nsample=50)

Hamming (0-NumAgents)= NA

Increments: (mean=4955.5, sd=214.4, sterr=30.32, min=4471, max=5355, Nsample=50)

Test 7: Hamming Target after CPU Load Balancing and Loss of Two Hosts

As an example of using the Hamming target, this case will use the state of the system after a CPU balancing strategy, using test case 1.1 as an example of the "target state" used for our hamming test. This hamming state, "a1.lb-loadbalcpu.hamming" is used as input to the run. We then disable servers Fwd-A and Fwd-B and see how close we come to the original deployment.

INPUT:

Target State: a1.lb-loadbalcpu.hamming

Disable Fwd-A and Fwd-B

hammingCheck = True

MinUltralogNodes = 1

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

Avoids FWD-B (pfail=0.8) and FWD-A (pfail=0.5), and then uses all other nodes, with the Hamming strategy among the remaining used nodes.

OUTPUT FILES: a1.lb-hamming-5host.state, a1.lb-hamming-5host.hamming

Server Summary:

Fwd-A				
Fwd-B				
Fwd-C	60.0% cpu	70.3% mem	2.4 risk	12 agents
Fwd-D	60.0% cpu	70.3% mem	2.4 risk	12 agents
Fwd-E	70.0% cpu	82.0% mem	2.8 risk	14 agents
Fwd-F	40.0% cpu	93.8% mem	1.6 risk	16 agents
Enclave2 Manager	100% cpu	55.9% mem	0.2 risk	2 agent

Solution Parameters:

Load Balance = 0.980

Remote Traffic = 626

Agent Risk (0-NumAgents)= 2.8

State Failure Rate (0-1) = 0.586

Hamming (0-NumAgents) = 16 (16 min, 23 max)

Increments = 3,387

Objective = 5.71 E-1

Test 8: CPU Load Balancing after Loss of Two Hosts

Disable the two highest failure servers, Fwd-A and Fwd-B, and ask the solver to find a CPU-balanced solution on the remaining 5 hosts. Compare this to the min state failure solution (Test 2), and the previous Test 7 solution based on finding a hamming solution, or closest solution to previous state before loss of hosts Fwd-A and Fwd-B.

INPUT:

Fwd-A, Fwd-B = disabled

Set Load Balancing CPU/Memory = 1.0 (pure CPU balance with memory constraints)

Strategy = Load Balancing of CPU and Memory

loadbalCheck = True

loadbalcpumemratio = 1

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

Avoids FWD-A (pfail=0.5) and FWD-B (pfail=0.8), and then uses all other nodes, with a cpu balance performed on the remaining used nodes.

OUTPUT FILES: a1.lb-loadbalcpu-5host.state, a1.lb-loadbalcpu-5host.hamming

Server Summary:

Fwd-A				
Fwd-B				
Fwd-C	60.0% cpu	70.3% mem	2.4 risk	12 agents
Fwd-D	65.0% cpu	76.2% mem	2.6 risk	13 agents
Fwd-E	65.0% cpu	76.2% mem	2.6 risk	13 agents
Fwd-F	42.5% cpu	99.6% mem	1.7 risk	17 agents
Enclave2 Manager	50.0% cpu	50.0% mem	0.1 risk	1 agent

Solution Parameters:

Load Balance = 0.444 (0.444 min, 1.17 max)

Remote Traffic = 549

Agent Risk (0-NumAgents)= 2.6

State Failure Rate (0-1) = 0.586

Hamming (0-NumAgents) = NA

Increments = 5,416

Objective = 3.17 E-2

COMMENTS:

Notice that the load balancing objective has a value of loadbal = 0.444 in this test, compared to the value of loadbal = 980 in the previous test. This is to be expected, since in test 7, we are using the hamming objective as our objective function, whereas we are using the actual value of loadbal as our objective.

Test 9: Mixed Strategy: Min State Failure + CPU Load Balancing

This is an automated mixed strategy that replicates the test performed in the previous test (Test 8: CPU Load Balancing after Loss of Two Hosts).

INPUT:

Set “Blend Pfail and Load Balance” flag in annealing form to flag two pass run

Set Load Balancing CPU/Memory = 1.0 (pure CPU balance with memory constraints)

blendpfailloadbal = True

loadbalcpumemratio = 1

MinUltralogNodes = 1

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

Avoids FWD-A (pfail=0.5) and FWD-B (pfail=0.8), and then uses all other nodes, with a cpu balance performed on the remaining used nodes.

OUTPUT FILES: a1.lb-2pass-minfail-loadbalcpu.state, a1.lb-2pass-minfail-loadbalcpu.hamming

Server Summary:

Fwd-A

Fwd-B

Fwd-C 65.0% cpu 76.2% mem 2.6 risk 13 agents

Fwd-D	65.0% cpu	76.2% mem	2.6 risk	13 agents
Fwd-E	60.0% cpu	70.3% mem	2.4 risk	12 agents
Fwd-F	42.5% cpu	99.6% mem	1.7 risk	17 agents
Enclave2 Manager	50.0% cpu	50.0% mem	0.1 risk	1 agent

Solution Parameters:

Load Balance = 0.444 final pass (0.444 min final pass, 1.15 max final pass)

Remote Traffic = 623

Agent Risk (0-NumAgents)= 2.6

State Failure Rate (0-1) = 0.586 final pass (0.586 min first pass, 0.959 max first pass)

Hamming (0-NumAgents) = NA

Increments = 8,696 (2 pass)

Objective = 3.17 E-2

COMMENTS:

The results of this test and the previous test should be close, since they are solving the same mathematical problem. In the previous test (test 8), the two high failure rate nodes are manually disabled, whereas in this test, the two high failure nodes are automatically disabled in the first pass of this two pass mixed strategy. The results should be close in the two objective function components that are targeted (State Failure Rate, Load Balance). The agreement between test 8 and this test in the Agent Risk component is purely by chance, since this is not a targeted dimension of the objective function. The disagreement in the remote traffic component is to be expected, since Remote Traffic is not a targeted dimension of the objective function. The flip-flop in server summary statistics between Fwd-C (60% CPU test 8, 65% test 9) and Fwd-E (65% CPU test 8, 60% test 9), is also not important, since the objective function is sensitive to the overall load balance, and is not sensitive to how this load balance is achieved.

Test 10: Mixed Strategy: Min State Failure + Min Messaging

INPUT:

Set “Blend Pfail and Messaging” flag in annealing form to flag two pass run

blendpfailmessaging = True

MinUltralogNodes = 1

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

Avoids FWD-A (pfail=0.5) and FWD-B (pfail=0.8), and then uses all other nodes, with a Minimize Remote Messaging strategy performed on the remaining used nodes.

OUTPUT FILES: a1.lb-2pass-minfail-minmsg.state, a1.lb-2pass-minfail-minmsg.hamming

Server Summary:

Fwd-A			
Fwd-B			
Fwd-C	85.0% cpu	99.6% mem	3.4 risk 17 agents
Fwd-D	85.0% cpu	99.6% mem	3.4 risk 17 agents
Fwd-E	85.0% cpu	99.6% mem	3.4 risk 17 agents
Fwd-F	7.5% cpu	17.6% mem	0.3 risk 3 agents
Enclave2 Manager	100.0% cpu	55.9% mem	0.2 risk 2 agents

Solution Parameters:

Load Balance = 1.65

Remote Traffic = 232 final pass (232 min final pass, 733 max final pass)

Agent Risk (0-NumAgents)= 3.4

State Failure Rate (0-1) = 0.586 final pass (0.586 min first pass, 0.959 max first pass)

Hamming (0-NumAgents) = NA

Increments = 7,675 (2 pass)

Objective = 4.64 E-2

Test 11: Additional Hamming Tests (Multiple Tests)

For each of the previous runs in this set of tests, write out a hamming target file to "projectname.hamming". You will need to copy the individual test case output files to a special name, since the default write is written to the same "projectname.hamming" and "projectname.state" file each time. For example, we might rename files after each run to names like "projectname.test-1.hamming", and then rename the file back to "projectname.hamming" before reading the file in as a target hamming file.

Subtest 1:

For each test in the above set, read in the Hamming output file as a Hamming input file for the new run. Select the "Hamming" option for the optimizer (no other switches selected). Set the MinUltralogNodes = 1. Leave all other options identical to the run that produced the target Hamming file (i.e., function/server eligibility, etc.), so that the target server/link utilizations are identical to the previous run that generated the Hamming target.

Expected Result:

In each case, the solver should produce an output solution that is identical to the input solution.

Subtest 2:

For each test in the above suite, read in the Hamming target for a new run, and break one or more of the key components that are used in the run.

Expected Result:

The model will try to find the closest fit. For example, if we break one of the nodes with five agents on it, then the best we can do is find a solution five or more Hamming steps away. Note that a pure hamming run ignores other measures, like survivability.

A truly blended objective function gets us into the area of multi-objective optimization, where tradeoffs can be a bit fuzzy. For this integration period, we are considering only two pass iteratations for fused objectives, where the first pass is minimization of system failure. The second pass can be Min Hamming Distance, OR Max Survivability, OR Min Messaging:

Fused Two Step Strategies:

First Step is always the minimize system failure strategy. In the first pass we discard high risk nodes from the search.

1.) “Min State Failure” and “Min Remote Messaging”

2.) “Min State Failure” and “Min Agents at Risk” (Max Survivability)

3.) “Min State Failure” and “Min Hamming Distance to Another Target State”

Test 12: Mixed Strategy: Load Balancing (CPU/Mem) + Remote Messaging

INPUT:

Set “Blend Load Balance and Messaging” flag in annealing form to flag two pass run

Set Load Balancing CPU/Memory = 1.0 (pure CPU balance with memory constraints)

blendloadbalmessaging = True

loadbalcpumemratio = 1

MinUltralogNodes = 1

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

All nodes are used, with a mixture of the two objectives of load balancing of CPU and min messaging.

OUTPUT FILES: a1.lb-2pass-loadbalcpu-minmsg.state, a1.lb-2pass-loadbalcpu-minmsg.hamming

Server Summary:

Fwd-A	40.0% cpu	23.4% mem	4.0 risk 8 agents
Fwd-B	30.0% cpu	17.6% mem	4.8 risk 6 agent
Fwd-C	40.0% cpu	46.9% mem	1.6 risk 8 agents
Fwd-D	45.0% cpu	52.7% mem	1.8 risk 9 agents
Fwd-E	45.0% cpu	52.7% mem	1.8 risk 9 agents
Fwd-F	37.5% cpu	87.9% mem	1.5 risk 15 agents
Enclave2 Manager	50.0% cpu	50.0% mem	0.1 risk 1 agents

Solution Parameters:

Load Balance = (0.262 first pass, 0.405 final pass) (0.262 min first pass, 1.82 max first pass)

Remote Traffic = (670 first pass, 231 final pass) (207 min final pass, 737 max final pass)

Agent Risk (0-NumAgents)= 4.8

State Failure Rate (0-1) = 0.959

Hamming (0-NumAgents)= NA

Increments = 7,261 (two passes)

Objective = 4.62 E-2 final

COMMENTS:

This mixed strategy is accomplished by performing a CPU load balance in the first pass, and then trying to minimize remote messaging in a second pass. The second pass objective function uses the standard remote messaging objective function, but with a penalty factor that penalizes solutions that have a load balancing component that is more than 10% above the minimum load balancing objective found in the first pass. Notice that in this example, the first pass load balancing objective ranges from 0.262 to 1.82, for a range of 1.558. This means that our exponential damping factor in the second pass begins at 0.262 + 0.1558 = 0.4178. Notice that the final load balancing component in the final pass is 0.405, which is below 0.4158, so the penalty does not contribute. The remote traffic component is 231, which is towards the bottom of the range of 207 to 737. This is a good compromise between the conflicting objectives of CPU load balance and minimum remote messaging. It is important to note that a final pass with a load balance above the start of the penalty region (i.e., loadbal > 0.4178) would also be acceptable, although the exponential penalty keeps loadbal close to the non-penalized region (i.e., loadbal < 0.4178). An example of this occurred in another trial of this problem, where the final solution had a load balancing value of loadbal = 0.640, and a value of remote traffic component of remotetraffic = 191. In this second example, the value of loadbal is clearly inside the penalty region, although it still shows an acceptable value of load balancing, while the remote traffic component is better than that seen in the example above. These varied results should be expected for any non-deterministic optimization procedure which searches for good solutions (as opposed to the global optimum), and also should be expected in any optimization procedure which deals with conflicting objectives (i.e., the mixed strategies).

Test 13: Soft Constraints with Min Remote Messaging

In this test, we attempt a Min Remote Messaging strategy, with a set of servers that are clearly insufficient to support the agent load. We manually disable Fwd-A, Fwd-B, and Fwd-C. After a first pass that attempts the Min Remote Messaging strategy, the solver automatically switches to the CPU+memory load balancing strategy with soft constraints for a second pass. The CPU and memory load balancing are equally weighted (loadbalcpumemratio = 0.5).

INPUT:

Strategy = Minimum Remote Messaging

Fwd-A, Fwd-B, Fwd-C = Disabled

MinUltralogNodes = 1

remotetrafficCheck = True

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

Fwd-A, Fwd-B, and Fwd nodes are avoided, with a load balancing of CPU and memory on the remaining nodes.

OUTPUT FILES: a1.lb-softconstraints-4host-minmsg.state, a1.lb-softconstraints-4host-minmsg.hamming

Server Summary:

Fwd-A				
Fwd-B				
Fwd-C				
Fwd-D	85.0% cpu	99.6% mem	3.4 risk	17 agents
Fwd-E	85.0% cpu	99.6% mem	3.4 risk	17 agents
Fwd-F	50.0% cpu	117.2% mem	2.0 risk	20 agents
Enclave2 Manager	100.0% cpu	55.9% mem	0.2 risk	2 agents

Solution Parameters:

Load Balance = 0.825 (0.825 min, 7.24 max)

Remote Traffic = 535

Agent Risk (0-NumAgents)= 3.4

State Failure Rate (0-1) = 0.482

Hamming (0-NumAgents)= NA

Increments = 10,966 (two passes)

Objective = 5.89 E-2 final

COMMENTS:

All pure and mixed strategies are automatically converted to load balancing of cpu and memory with soft constraints after failure to find a solution with hard constraints. This example with four hosts had a problem satisfying the memory constraints. Although this test used the min remote messaging strategy as the initial strategy, any other strategy should exhibit similar results.

Test 14: Soft Constraints with Load Balancing of Risk

In this test, we attempt a Load Balancing of Risk strategy, with a set of servers that are clearly insufficient to support the agent load. We manually disable Fwd-A, Fwd-B, Fwd-C, and Fwd-D. After a first pass that attempts the Load Balancing of Risk strategy, the solver automatically switches to the CPU+memory load balancing strategy with soft constraints for a second pass. The CPU and memory load balancing are equally weighted (loadbalcpumemratio = 0.5).

INPUT:

Strategy = Load Balancing of Risk

Fwd-A, Fwd-B, Fwd-C, Fwd-D = Disabled

MinUltralogNodes = 1

pfailuresurviveCheck = True

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

Fwd-A, Fwd-B, Fwd-C and Fwd-D nodes are avoided, with a load balancing of CPU and memory on the remaining nodes.

OUTPUT FILES: a1.lb-softconstraints-3host-agentrisk.state, a1.lb-softconstraints-3host-agentrisk.hamming

Server Summary:

Fwd-A				
Fwd-B				
Fwd-C				
Fwd-D				
Fwd-E	140.0% cpu	164.0% mem	5.6 risk	28 agents
Fwd-F	62.5% cpu	146.5% mem	2.5 risk	25 agents
Enclave2 Manager	150.0% cpu	61.7% mem	0.3 risk	3 agents

Solution Parameters:

Load Balance = 1.260 (1.260 min, 11.40 max)

Remote Traffic = 303

Agent Risk (0-NumAgents)= 5.6

State Failure Rate (0-1) = 0.353

Hamming (0-NumAgents)= NA

Increments = 27,237 (two passes)

Objective = 8.99 E-2 final

COMMENTS:

All pure and mixed strategies are automatically converted to load balancing of cpu and memory with soft constraints after failure to find a solution with hard constraints. This example with three hosts had a problem satisfying both the CPU and memory constraints. Although this test used the load balancing of risk strategy as the initial strategy, any other strategy should exhibit similar results.

Test 15: Soft Constraints with Load Balancing of Risk

In this test, we attempt a Load Balancing of Risk strategy, with a set of servers that are clearly insufficient to support the agent load. We manually disable Fwd-A, Fwd-B, Fwd-C, Fwd-D, and Enclave2 Manager. The problem is further complicated by the fact that the agent "Enclave2ScalabilityManager" is only eligible to run on the node Enclave2 Manager. The CPU and memory load balancing are equally weighted (loadbalcpumemratio = 0.5).

INPUT:

Strategy = Load Balancing of Risk

Fwd-A, Fwd-B, Fwd-C, Fwd-D, Enclave2 Manager = Disabled

MinUltralogNodes = 1

pfailuresurviveCheck = True

INPUT FILES: a1.nodedef, a1.functiondef, a1.linkdef

OUTPUT:

No Solution Found.

OUTPUT FILES: N/A

Server Summary:

Fwd-A

Fwd-B

Fwd-C

Fwd-D

Fwd-E

Fwd-F

Enclave2 Manager

Solution Parameters:

Load Balance = NA

Remote Traffic = NA

Agent Risk (0-NumAgents)= NA

State Failure Rate (0-1) = NA

Hamming (0-NumAgents)= NA

Increments = NA

Objective = NA

COMMENTS:

While the soft constraint mode of the load balancer is designed to deal with problems in resource availability, it is not designed to perform magic. Since the “Enclave2ScalabilityManager” agent has no eligible servers for execution, then no solution is possible. In this case, the load balancer returns the standard “solution not found” flag, which is a probability of failure = -1 (minus one)

11 Appendices: HPC 2004 Conference Paper

This appendix includes the paper that was presented to the “High Performanc Computing Conference” in Cetraro Italy. The talk was given on June 2, 2004, and summarized our Load Balancing efforts and suggested enhancements to improve performance of distributed agent systems.

Performance and Risk in Large Distributed Systems: Case Study of DARPA Ultralog

Marc Brittan

Boeing Mathematics and Computing Technology, P.O. Box 3707, Seattle Washington, 98124, USA

Abstract:

Management of performance and risk in distributed computing has been a problem since the beginning of this technology. While the problems in risk and performance management are known to be intractable (NP-Complete, #P-Complete), recent advances in both computer speed and algorithms have opened up new opportunities in distributed system design. Modern systems are now capable of addressing near-real-time systems management problems that were previously considered infeasible. In this paper we discuss our experiences in the DARPA Ultralog project, and propose significant extensions to the state management architecture that should improve our abilities to dynamically manage performance and risk in distributed systems. The paper also discusses issues in constraint satisfaction and thread modeling that are used to build near-real-time performance models. Since speed is imperative for real time risk and performance management, the paper discusses the connection between effective selection of performance threads for performance modeling, and the design of effective state moves used by the solver in searching the solution space.

Keywords: Agents, NP-Complete, Survivability, Distributed System Performance

Corresponding Author: Marc Brittan, Marc.Brittan@Boeing.com

1. Introduction:

Problems in performance, risk, and resource management have been at the core of distributed processing since the advent of this technology. A key problem in building performance and risk management tools into distributed operating systems is that the classic problems in distributed system design, like problems in queuing theory, combinatorial optimization, cost, and failure analysis, must now be confronted by the operating system in near real time. One of the things that prolongs the design phase for large distributed systems is that these problems are combinatorial in nature (Job/server/processor assignment, routing, Number of CPU's per host, etc.), and classically intractable (NP-Complete, #P-Complete). This complicates the problem of incorporating performance and risk management into an operating system, since the underlying problems in distributed design are themselves computationally expensive. Given the significant advances in processor speed and algorithms in recent years, we have reached a point where many hard problems in distributed system design can be solved in near real time for large systems. This ability to model real world systems in near real time opens up new opportunities in dynamic performance and risk management of a distributed system, and new opportunities to address existing problems in a distributed and parallel environment.

To take advantage of real time performance and risk management, we also need a flexible architecture that allows the jobs and workloads to be moved around the system. This mobile architecture is needed to both take advantage of underutilized resources, and to take advantage of lower risk services in a high risk environment. In this paper we will present some of the techniques being investigated in the DARPA Ultralog project on survivable distributed systems. The Ultralog program is currently focused on logistics applications, although the architecture is generally applicable to most computing problems.

It is common for large problems in distributed processing to take days to complete, and in some cases (like large military campaigns) weeks or months to complete, so robust distributed computation is vital. The prospect of having a component failure force a restart of a lengthy calculation means that it is important to periodically generate checkpoint data for a process or task to ensure that our restarts of failed processes are based on recent states of the system.

In DARPA Ultralog, the computing tasks are performed by a set of mobile programs or agents. The agent architecture that is being used is the Cougaar agent architecture (www.Cougaar.org), and was originally developed to deal with large scale logistics applications. The agents are allowed to move their processing to any eligible server on the network. It is the job of the state manager (topic of this paper) to describe how this workload is managed to improve performance and reduce risk.

We will define a ***state configuration*** (or simply "state") as a set of agent/server assignments, and the required routing tables to support the inter-agent messaging. It is the job of the state manager to choose an operating state for the system that optimizes various measures of risk and performance. In deployment, the Ultralog system is a large ***society*** of agents, with the overall society broken down into smaller groups called ***enclaves***. We are targeting a typical enclave for our state optimizer of about 150 agents on 75 servers. On this size of problem our solver for the state optimizer returns a good solution in 10 seconds or less on an 800 mhz Linux pc. This solver response time has been more than adequate, given the time it takes to move agents to new servers. This run time (seconds) is useful at the small network level up to a class C (255 hosts) with a few hundred agents. Because of this size limit for state management, our current approach is to break the larger Ultralog society into multiple smaller subnetworks, and manage performance and risk at the subsystem level.

2. Operational Modes and Defense Strategies:

Information on resource use and performance, along with threat and risk information from the user community, is used by a controller agent to select a strategy or mode of operation for the system. These strategies are designed to minimize risk or improve performance of the system by placing the system into a particular configuration.

The Ultralog system is naturally divided into enclaves with each enclave managed in its own local environment. If the risk level for an enclave is perceived by the controller to be high, then the controller agent would have the state optimizer find a state that minimizes risk. In a high risk situation the default procedure is to reduce risk using a two-pass procedure in the state optimizer. In the first pass the solver removes high risk nodes and links in the system (find the safest set of hosts/links), which minimizes the probability of state failure from all causes. Since a break in a component either forces agents to move, or routes to change, then we try to minimize overall state failure in the first pass of a risk based strategy. In the second pass of a high risk strategy, we spread the agents around the remaining servers, and level out the risk profiles for each server. In this second pass, our goal is to minimize the maximum expected damage from a single point attack. This approach places more agents on low risk servers, and fewer agents on high risk servers. This mixed risk strategy minimizes the overall failure rate for the state, and then levels out the risk profiles on the remaining servers.

If an enclave was at risk for corrupted communications, then the controller might choose another strategy for that part of the system, and operate that particular subsystem under a minimize remote messaging strategy (described below). In this case, the solver will find a state that reduces traffic between servers, and places agents with high rates of communications on the same servers. Still another strategy might be chosen by the controller if a LAN segment appeared to be low risk. In this case, the controller might choose a standard load balancing strategy to improve performance for the enclave.

3. Agents and Distributed Computing

Agent based systems have become increasingly popular over the last few years, and are now being deployed in a variety of applications in commercial and scientific work. While relocatable tasks and load balancing have been around for many years, the development of agent technology has formalized this task, and made the process of building an agent system accessible for both common and exotic processes. The networked agent system brings with it the potential for parallelism, and autonomy, and of course the headaches of performance management in a network where everything moves during processing.

The use of mobile agents in the design of a distributed calculation now serves as an effective complement to our adaptive routing methods that have been in use for years. While networks have long had the ability to adjust to damage to communications links by “routing around the damage”, our systems typically have not had this ability to readily move jobs around the system. The DARPA Ultralog architecture is in a sense the marriage of a flexible software architecture to a flexible hardware architecture, with jobs and messages moving and adjusting in real time to performance needs and system risks. Of course, this puts new demands on our state manager, since it must obtain resource, usage, and risk information for an entire enclave, and solve a number of NP-Hard problems in system design using this data.

Although the distributed system design problem has many NP-Complete sub-problems, it is relatively easy in a resource-rich environment, where there is large server power and bandwidth compared to the job and messaging requirements. A simple greedy search usually works well for system design problems with abundant resources. The full NP-Completeness (and difficulty) of the problem is felt in resource-marginal situations, where there is little excess capacity in the system. This is where techniques in simulated annealing, genetic algorithms, and other specialized search techniques are needed to find good solutions in reasonable time. Since the UltraLog goal is to build a system to withstand 45% destruction, then we must be prepared for “resource-marginal” situations.

Note that this flexible architecture also has implications for security, since we are now presenting our attackers with a moving target. By continually changing task/server assignments, ports, protocols, and other parameters, our attackers must now try to crack into a system before the system changes its configuration, which creates a complex moving target for attackers.

Of course, this new computing model comes at a cost. One of the costs for this implementation is that our agents must generate checkpoint information for their internal process state, and store this information on a remote host for a possible restart after an attack or system crash of the current host. In Ultralog, the checkpoint information is currently stored on a single remote host, although investigations are underway to move the system into a distributed checkpoint mode of operation. The actual process of generating and storing checkpoint data for states of the system can be extremely involved, and is at least NP-Hard when the goal is to design a system with distributed persistence.[1]

4. Agent Messaging:

One of the common design goals in building healthy distributed systems is to minimize remote messaging between all servers. This approach is frequently used in distributed database design, and uses a technique in mathematics known as graph partitioning, which is NP-Complete [2]. Graph partitioning can be an effective tool for improving response time and performance in situations where the inter-host messaging time is significant compared to processing time at the servers. This strategy may also be selected by the defense controller when it is perceived that there are significant errors in transmission, or when the system is trying to enhance security by minimizing the remote messaging for an enclave. The objective function for the graph partitioning problem is given below, where the sum is taken over all agent pairs that are assigned to different servers:

Minimize Remote Messaging Strategy (graph partitioning):

$$\text{Minimimize} \quad \{ \sum_{\text{Agents}} \text{Msg}(i, j) \} ,$$

where agents i , j are in valid states on different servers

A common criticism of graph partitioning is that the minimization of remote calls does not directly address response time. Response time is measured by tracking the time-length of all parallel threads spawned by an initial event, such as a user query. Clearly there can be many situations where we minimize remote messaging on the system and increase response time. We will discuss response time in greater detail in a later section.

5. Server Health and Constraints

In many cases we lack detailed knowledge of how the system will be used, and are limited to estimates of current use of CPU and memory for an agent. We can use this limited information to find system designs that are in the *healthy* operational limits of the system's components. The assignment problem of agents to servers such that CPU and memory constraints are satisfied is a variant of the two dimensional bin packing problem and is NP-Complete [2].

In general, we try to keep resource utilization levels well below 80%, which is typically the beginning of the “knee-of-the-curve” in M/M/c queuing theory - the point where long lines of jobs/tasks start piling up in queues, waiting for service at an overloaded resource. When the CPU Utilization levels are high (e.g., above 80%), we see large increases in user response time due to queuing at the CPU processors. CPU is treated as a constraint in all Ultralog strategies, and is only included in the objective function for pure load balancing strategies based on CPU load balancing.

In our current memory model, we calculate the total memory consumption on a given host in the system by adding the memory requirements for each agent/program on that server. Our current Ultralog model does not treat shared memory, although that would be a useful extension that would not increase the overall computational complexity of the problem (already NP-Hard, #P-Complete). We would like our servers to have enough memory to keep the agents in their healthy zones of operation. When an agent or program is starved for memory, it means that some pieces of data or code must be read from disk and saved to disk (page faults and virtual RAM). This disk retrieval is a much slower process than reading from RAM.

6. Response Time

In a response time model of a system, we model the sequences of calls generated by an initial function call or agent task. The response time is defined as the time-length of the longest running thread in the set of all threads generated by the initial call. For example, a user or agent might initiate a task that is handled by a variety of agents performing subtasks. Each of these subtask agents may themselves have subtasks, thus forming a series of threads/trees of agent tasks. We have not implemented thread tracking and optimization in UltraLog, since the availability of the real-time job arrival and job size information needed for thread tracking and queuing theory was not yet available.

The state optimizer for UltraLog was built by Boeing from the core of a larger and more complex program used for designing large distributed systems (Boeing has several extremely large distributed systems). This larger design program had a special emphasis on the intractable parts of the distributed system design problem which are difficult to deal with for large systems. The Boeing solver was automated to run in the UltraLog agent environment, and programmed to run in real time to manage performance and risk of an agent system. In the larger Boeing solver, the optimizer does track threads (with queuing theory corrections) generated from an initial user query for true response time, and does this thread tracking in a TCP/IP environment (with packetization). In this thread-based model, the solver searches out the design space to minimize the length of the longest thread generated by an initial user query or agent task [3]. The objective function for the Thread Response Time Strategy used in our internal Boeing tool is defined below, where we minimize the time-length of the longest running thread (thread tracking not performed in Ultralog). The threads are generated by the solver at run time from user-defined inputs that describe the performance threads and software calling architecture.

Objective: Minimize Thread Response Time (not in Ultralog)

Minimize { Max [T(i)] }

T[i] = Time-length of i’th thread in test set of threads

One of the most common techniques for estimating response time of a system is to build a simulation model of the system. To simulate a computer system, we have computers model artificial users generating artificial jobs at random. Each of these virtual jobs may then spawn new jobs, and generate a threaded set of calls that are tracked in the simulation. A simulation model frequently samples millions of artificial jobs submitted by thousands of artificial users in the simulation, and measures the average user response times along with other systems performance parameters like CPU utilization and memory. While this is the preferred modeling technique for complex random processes, it is also a computationally lengthy task. A simulation model of a distributed system might take hours to complete on a fast workstation for one specific state of the system (agent/server and routing). This might be useful for a careful detailed study of a system, but in our distributed design problem we may have an exponentially large number of possible design options. A typical agent system may involve a thousand agents in a complex network with hundreds of hosts. The large number of design options is a problem with classic simulation, since we may end up trying to simulate an exponentially large number of design options - not a good choice for real-time automated performance and risk management of a distributed system!

We can quickly prune this design space using a variety of heuristics, but the overall number of possible design options is still extremely large. While a global solver based on simulation would be ideal, it is not feasible with current processor technology, so we have developed a variety of fast approximations for the core solver. To get a fast first estimate of a system design, including queuing effects associated with random job arrivals and random job sizes, we have been using fast analytic estimates from queuing theory to build queuing networks that model user transactions. This combination of fast analytic queuing plus global optimization based on simulated annealing with fast heuristics has allowed us to build a near real time tool for distributed agent management.

7. Performance Threads

A performance model is built from a set of threads of interest in a larger system. The choice of performance threads is quite complex in the general case, although special cases like classic three or four tier architectures can be modeled with a few basic threads to represent types of user transactions. For the objective function we calculate the response time for each thread in the set of thread samples, and sum the weighted thread times.

In a medium to large systems we encounter the usual NP-Hard problems that plague the distributed system performance industry. To deal with this need for improved thread modeling for real time tools, we have developed a variety of heuristics for thread selection, and heuristics for state transitions involving threads. The selection of threads frequently involves identifying a set of “user centers” or places where agents may initiate tasks. If we are not tracing the entire transaction (i.e. not tracing all threads generated from the initial task), then we select a set of sub-threads which we believe may be dominating the response time for tasks of interest.

The selection of sub-threads for a performance model may involve looking at the traffic matrix for inter-agent messaging, and identifying compute intensive or message intensive threads buried in the overall set of threads. Once we have selected a set of threads for our performance model, the next step is to build these threads into our objective function, and choose a set of state transitions to be used by the optimizer. The key, as with most optimization problems, is to structure the solver so that it captures the essence of the problem. In a later section we will describe a basic set of moves in the solution space that help the solver find good solutions in reasonable time.

8. Modeling Risk

The UltraLog system is designed to self-adjust when it detects a threat, and move jobs and communications to protect the system. One of the primary tasks of our automated performance/risk manager is to propose designs that minimize failure, since failure of a component forces agents to move, which delays processing. In a high risk situation we may want to eliminate the server from active service, or deploy fewer agents on that server.

To minimize state failure and other risk measures in UltraLog, we implement a mixed strategy that initially eliminates high risk nodes and links from use, and then minimizes the agents at risk from a single point attack on the remaining nodes. We currently assume that failures in the system are statistically independent. The objective function for the first phase of this two pass mixed strategy is the probability that one or more component failures occurs anywhere in the set of components supporting the state. This objective is calculated by first calculating the probability that there are no failures, which is just the standard product, over all physical components in the state, of the probability of that component being operational. This is the standard "uptime" calculation for the state of a system, and will tend to eliminate servers and links from the design (unless constrained), since they represent risk. In this first phase, called the "Minimize State Risk" strategy, we are reducing the risk of the system to some acceptable minimum, and minimizing our exposure to risky components that can break the current state of the system.

The objective function for the second phase of our mixed risk strategy, called the "Minimize Expected Single-Point Risk" strategy, is given below. In the actual implementation, we perform a leveling of risk among the servers after minimizing the maximum expected single-point risk, producing a smoother risk profile for the system:

Minimize Expected Single-point Risk Strategy:

$$\text{Minimize} \left\{ \text{Max}_{(\text{inod} \in \text{system})} \left[\text{Pfail}(\text{inod}) * \text{NumAgents}(\text{inod}) \right] \right\}$$

Pfail(inod) = Probability of failure for node number inod

NumAgents(inod) = Number of Agents on node number inod

Note that we have described a number of objectives for the state management problem, and in most cases these are conflicting goals. Since the problem is intrinsically a problem in multi-objective optimization, this means there are tradeoffs. The state optimizer currently offers a variety of pure strategies, like "Minimize State Risk", "Minimize Remote Messaging", and "Minimize Expected Single-Point Risk", and some mixed strategies, like our previously described two pass strategy for Minimize State Risk followed by Minimize Expected Single-Point Risk.

9. Building the Solver, and Solver Self-Attack:

One of our key design goals in DARPA Ultralog is to design a system that can withstand damage to 45% of the system infrastructure, and suffer no more than 30% performance degradation. To place this design goal into a real time system that reconfigures itself means that we must address a number of distributed design problems in near real time. A distributed system design model, or models, must deal with a number of NP-Hard problems in optimization and constraint satisfaction that are mixed with classic hard problems in queuing, cost modeling, and failure and reliability analysis, with the reliability problem known to be #P-Complete. Since we cannot possibly generate a full reliability study with all possible solutions in *real time*, we have designed a fast approach which studies the solution space and performs a series of smart attacks on the system to find key weaknesses in near real time.

Since a major part of the Ultralog goal is to "withstand damage to 45% of the system infrastructure and suffer no more than 30% performance degradation" then a natural question is "which 45% should we destroy, before testing performance?". Clearly we cannot search over all possible failure combinations (up to 45% of system), since that is exponentially large. To get a fast estimate of system hardness, we have developed an attack plan that uses sampling from our solver to estimate attack points on a system. The attack is based on a sequence of attacks that terminates after the system fails to find a viable state (i.e., performance has degraded by more than 30%). At each point in the attack sequence, a single physical component involved in supporting the current state is mathematically broken in the model. Some of the components that support the current state will occur in many other high quality states of the system (e.g., well connected routers), while other components (e.g., small servers) may only be involved in a few possible states of the system. At each point of the attack we attempt to choose the single component in the state that has the highest quality-weighted frequency of occurrence in the set of all failover plans for the system. By breaking this component, we break the current optimal state at the physical point in the state that also breaks the greatest number of high quality failover plans. This attacks both the optimal solution, and performs a semi-greedy attack on the failover system. Once we have mathematically broken a component, we find the new optimal state of the system, and then plan a new attack. This process of planning an optimal state, planning an optimal attack, planning a new optimal state, planning a new optimal attack is used as a hardness test of the system.

The attack part of the algorithm uses the solver to run a sensitivity analysis on the solution space, looking for physical components that occur in a large number of possible states of the system. Since the solver samples from tens of thousands to millions of possible solutions to the distributed system design problem, then we can use information from the sample set to estimate points of attack on the system. While the solver is busy looking for an optimal solution, it is tracking the frequency of component use for each sampled state of the system. It uses this to build a real time "reliability and performance map" of the design space during the solution process. The actual map is a set of numbers, P($\mathfrak{I}$, *objectid*), representing the weighted frequency of occurrence of the individual servers, links, and routers in the set of "high quality solutions" sampled by the solver in the distributed agent assignment problem.

$\mathfrak{I}$ = Set of all sampled solutions found by the solver that meet hard constraints

P($\mathfrak{I}$, *objectid*) = Probability that random selection from $\mathfrak{I}$ contains ***objectid***

where ***objectid*** is a Link, Router, Server, or other component of physical topology

In a simulated annealing model the solution will converge at very low temperatures to the estimate of the globally optimal state. At high temperatures, the annealing solver will move randomly about the distributed system performance space, sampling a broad range of low quality to high quality solutions. At medium and lower temperatures, the higher quality states will be preferentially sampled. We use this preferential sampling to build our quality weighted estimate of component use in the space of all possible states of a system.

We are using the well known Metropolis algorithm [4, 5] for this particular annealing algorithm, in which we accept downhill moves with probability one, and uphill moves with a probability that is exponentially damped with respect to the change (degradation) in the objective function. The terms in the exponent for the change probabilities are the change ***DE*** in the objective function in moving from the current state to the proposed state, and the temperature ***T***, which is a measure of randomness in the annealing algorithm.

Metropolis Algorithm: **downhill moves always accepted ,**

uphill moves exponentially damped

P(State Change) ∝ exp(-DE/T) (uphill move probability for case DE > 0)

We do not have the computing power to calculate P(ℑ, *objectid*) in real time since it is #P-Complete, so we build fast real time estimates of the frequency that a physical component occurs in the set of all solutions sampled by the solver. For each component we are going to sum the number of times that component appears in the set of solutions sampled by the solver. We do this by dividing the cooling region in the annealing algorithm into several steps (10 in our solver) between a "hot" temperature *Thot* and the "cold" temperature *Tcold* where the system has converged to its estimate of the global solution. At each step of this cool down process, we calculate a quantity conceptually related to P(ℑ, *objectid*) , which we will label $P_{anneal}(\Im, objectid, T)$, which is the frequency of occurrence of an object in the set of solutions sampled by the annealing solver at the temperature T. While we cool the system during the annealing process, we build these counts for each physical component of the number of times that component has been used in a successfully sampled state. Once the search algorithm has cooled, we calculate the overall map of system usage as a simple average over the temperature regions sampled by our solver:

$$P\,\text{anneal}\ (\Im, objectid\ ) = \sum_{T=Tlow}^{Thot} P\,\text{anneal}\ (\Im, objectid\ , T)$$

We use this smoothing technique to estimate the component's weighted frequency of use in the sample set. The weighting is by solution quality, since higher quality solutions are sampled more often in the annealing algorithm. A map of the performance and risk solution space can be generated in near real time with this procedure. The links in the figure below are weighted according to the values of $P_{anneal}(\Im, objectid)$ generated by our solver in the above algorithm.

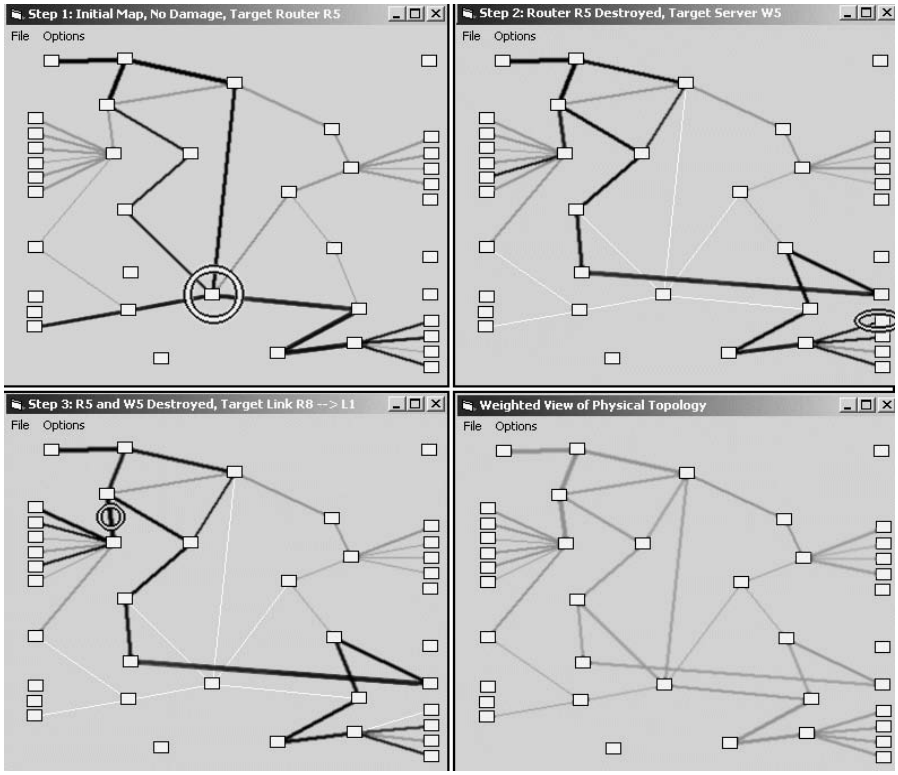


Figure 1: A sequence of four views of the solution space. The first map is for a fully operational system, followed by two maps generated after partial destruction of the system. The circles show where the next attack will take place. The optimal solution is in black, with backup solutions in gray. The white lines are for unused links that were used in previous steps. The fourth map represents a weighted summary of the solution space for distributed design.

In Figure 1 we have a series of four agent reliability and performance maps that illustrate views of the solution space seen by the solver. The first map (top left corner) of the system highlights the solver's best single solution in black, with backup/failover solutions drawn with gray lines. The thickness of the lines connecting the servers and routers is related to the frequency of occurrence of that link in the set of high quality solutions found by the solver. A similar view is possible (not shown) in which the servers and routers are drawn with sizes proportional to their frequency of occurrence in the solution set. A thick line occurs in a large number of quality solutions as sampled by the solver, whereas a thin line may belong to a few good solutions, but only a few solutions in comparison to the links drawn with thick lines. In addition to testing system hardness, the algorithm used to generate this map information can be used to design systems that have a large number of nearby high quality failover plans. This tends to minimize excessive agent movement after a failure, since the system has a number of nearby quality failover plans. Another positive aspect of this solver self-attack process is that it builds diversity into the real time solver.

Simulated annealing has been used for quite some time on a variety of difficult distributed design problems, such as the topological design of networks [6], and our own work on whole-system design by simulated annealing [3], in addition to other industrial applications. The technique requires care in designing moves about the search space, and in defining the annealing schedule [4,7], but once implemented the algorithm works well and is easy to modify. Annealing techniques are known for being slow at times (like all algorithms applied to NP-Complete problems), but can be accelerated with heuristics to become competitive with other techniques. Annealing algorithms are quite famous, however, for being robust, and we believe the technique is particularly well suited for this application (combinatorial optimization in a faulty queuing network), where the objective function is so complex that most classic optimization techniques fail.

Attempts to design reliable and robust networks have a long history. In 1964 Paul Baran from RAND considered several network architectures capable of withstanding atomic attacks. More recently Albert-Laszlo Barabasi and his collaborators [8] introduced the scale-free topology which preserves network connectivity after severe random damages. Havlin and his colleagues [9,10] proved that the scale-free topology networks are robust under certain circumstances. In future work we plan to consider the challenging task of combining topological robustness with the optimization approaches described in this paper.

10. Neighbourhood Structure in Simulated Annealing

A key problem in designing effective annealing algorithms is defining the neighbourhood structure of the search space. Our Markov chain in the annealing algorithm is based on moves from any given state to a neighbouring state. Defining this neighbourhood structure is a complex and application dependent problem. The application dependence of the annealing algorithm means that each new problem type (Traveling Salesman Problem, Graph Partitioning, Distributed System Design) presents new problems in moving about a solution space. For example, the classic k-opt moves of the Traveling Salesman problem are good examples of defining topologically close states that capture the essence of the problem, with a neighbourhood structure that allows efficient moves through the solution space [11]. For our distributed design problem, we are also going to define some basic moves through the solution space that capture the structure of the space and help the solver find quality solutions in a reasonable time.

One of the most basic state moves is to choose a single random host, and select a subset of jobs on the server to move to other random servers in the system. In this host selection step, we occasionally make heuristic state change proposals to the annealing solver. For example, in the mixed minimize risk strategy, a common step is to identify servers at high risk, and move agents from the high risk servers to the low risk servers.

By our definition of single state failure, the system will have failed regardless of whether a failed host had a single function running on it, or had all functions in the system running on it. Although we want our system to be capable of probabilistically exploring all possible task/host assignments, it is especially important for our failure analysis that the system be capable of reaching the empty server state (no running jobs) in a reasonable number of steps in the Markov chain. As a practical matter it is useful to design one of these moves to move *all* running agents on a given random server to other eligible servers in the system in a single step (or a few steps) of the Markov chain. This is a practical requirement for the failure component of the objective function, since we don’t want our state moves that add agents to a server to overwhelm the state moves that are trying to empty a server.

The next set of moves in our space is defined by selecting a random subset of functions/agents on a random set of hosts involved in a randomly selected thread in the set of all threads used by the performance model.

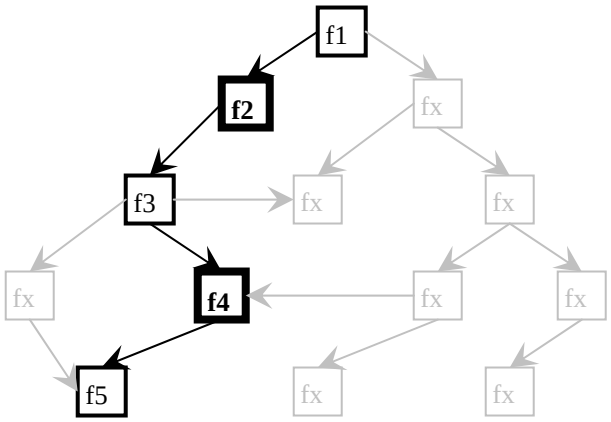


Figure 2: By selecting agents from a thread in a performance model, a set of state space moves are generated that capture the essence of the solution space. The selected thread is illustrated in black. A subset of functions (agents) on the selected thread are selected for movement to other servers. In this example the functions f2 and f4 (illustrated with bold black outlines) have been selected for movement.

In Figure 2 we have highlighted the selection of a single random thread of function calls embedded in a complex calling sequence initiated by an initial function or agent f1. The steps in our process for defining these threaded state moves are defined below:

1.) Select a single random thread $\mathcal{T}$ in the system. The thread $\mathcal{T}$ is a sequence of ordered pairs listing the functions and hosts in the thread, with [f(1), H(1)] being the first function or agent call in the threaded set of calls, and hosted on host H(1).

$\mathcal{T}$ = { [f(1),H(1)] , [f(2),H(2)], ...[f(n),H(n)] }

where:

f(i) = i'th function call in thread

H(i) = host associated with i'th function call in thread

2.) Select a random subset $\mathcal{T}sub$ of function/host pairs in $\mathcal{T}$.

3.) For each member of $\mathcal{T}sub$ randomly change the function/server assignment for each function in the subthread. For example if [f(i), H(i)] is in $\mathcal{T}sub$, then randomly change H(i) to one of the other eligible servers capable of hosting function f(i). As an option, randomly move one or more of the functions on H(i) (other than f(i)) to other servers, leaving f(i) fixed, which gives function f(i) more CPU and Memory on host H(i)), and changes the thread environment, which is a key requirement of a good state move. When thread-based response time is formally part of the distributed system design goal, then this thread-based state change is an effective means of moving about the performance solution space, and effectively captures the nature of the problem in the set of state moves.

For optimizing response time, these basic threaded state moves are an especially good set of state moves, since the overall response time of a job is equal to the length of the longest running thread in the set of threads generated by the initial task. By focusing on correlated state moves associated with calls within a thread, we are more likely to quickly remove any large scale design errors within a thread in a reasonable number of steps by the solver.

In our basic move, we take a specific thread of calls, and move a selected set of agents in the thread to other servers. This state move does a good job of covering the threadlike structure of our performance model. We have experimented with the use of smart state moves, like greedy heuristics for agent/server assignment in the annealing steps, and the preferential sampling of agents in a thread, and have found considerable speedup (at least a factor of 2), using an annealing algorithm based on careful selection of the state change operators compared to purely random state moves. This is to be expected, since the state change operators define the local neighbourhood structure of the solution space (topological closeness of one state to another state). By choosing our state moves to sample more heavily in the basic logical components of the problem (threads, risky servers, overloaded servers, etc.), we are able to get good solutions from an annealing algorithm in a reasonable amount of time.

It is possible to implement other basic state moves to improve solver speed for the state optimizer, like selecting a set of trees in the calling hierarchy, and moving selected agents on these calling trees to other servers. This tree-like basic step would help in situations with complex connectivity, since it allows annealing state moves that more closely match the treelike calling architecture of the thread tracking problem. It is also possible to generalize the threaded state move by selecting a set of threads { $\mathcal{T}i$ }, and performing multiple exchanges between threads as a basic state move. Since the solution space for a distributed system is extremely complex, anything we can do to improve the way we move about the space will improve the speed of the state optimizer. In this regard, annealing is similar to most algorithms, where a carefully designed and tuned algorithm that captures the nature of the problem can usually outperform other more general purpose techniques.

11. Conclusions

The DARPA UltraLog project has built and deployed a large scale experimental distributed agent system designed to improve system performance and survivability. The system is designed to self adjust to meet performance and risk goals and constraints in a changing environment. It is built on the Cougaar architecture of mobile agents that perform their tasks from multiple points on the network. The issue of performance and risk in an agent based system arises as it does with any distributed system, and is complicated by the autonomous and mobile nature of agents in a society. The natural parallelism present in many large scale problems makes them good candidates for an agent based solution with autonomous self-healing capabilities. The large logistics applications that are used as a test bed for DARPA Ultralog serve as testimony of the ability of large agent based systems to solve some of the world's most complex tasks in computing. Furthermore, the agent systems do this in a distributed and autonomous fashion, obtaining a solution speedup through parallel tasking in the agent environment, and improving survivability and security through the mobile agent architecture.

The design of a modern distributed system must make effective use of resources to build a system that meets capacity, response time, risk, and cost goals. The same intractable problems that we encounter in distributed system design are now being confronted by real-time solvers in the pursuit of real-time management of system performance and risk in a distributed agent environment. This real-time design problem (or redesign after attack or failure) is complicated by the frequently conflicting goals in performance versus survivability. In a survivable system, we try to spread the agents among several remote servers, so we do not have major parts of the system vulnerable to attack on any given host. From a performance perspective, we try to assign the agents to a few servers that are close together, or even one server, to minimize delays from remote messaging.

The UltraLog system is designed to be a highly flexible and robust computing architecture that can withstand attacks. In this system, the attackers will need to have considerable knowledge about the performance space if they are to have any hope of impacting the system. In the UltraLog system we have matched a flexible software architecture with a flexible hardware architecture to create a system that is survivable, testable, and meets performance demands in a distributed environment. In this paper we have described the techniques we have built into a tool to dynamically manage performance and risk of a large distributed agent system. We have also outlined some of the steps in the solution process, and some of the basic state changes, or basic moves in the solution space, that are used to improve solver speed for the state optimization problem.

In the process of building this system, we are confronted with some of the most difficult problems in computational mathematics, problems which must now be solved in near real time. The interesting blend of advanced mathematics, agent systems, and classic distributed system design is now being applied to one of our most critical problems in today's world of computing - the problem of designing systems that can self-configure themselves to improve performance, or self-configure to survive a smart attack.

References:

[1] S. Kraus, C. Tas and V.S. Subrahmanian, "Probabilistically Survivable Multiagent Systems", *Proc. 2003 Intl. Joint Conf. on Artificial Intelligence*, pages 789-795.

[2] Michael R. Garey and David S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*, New York: W.H. Freeman and Company, 1979.

[3] Marc Brittan, Janusz Kowalik, "Performance, Optimization, & Complexity in the Design of Large Scale Distributed Systems", *Parallel and Distributed Computing Practices*, Vol. 3, No. 4, Dec. 2000, Nova Science Publishers.

[4] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines*, New York: Wiley, 1989.

[5] S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi, "Optimization by Simulated Annealing", *Science* 220, 671-680, (1983)

[6] Cem Ersoy and Shivendra S. Panwar, "Topological Design of Interconnected LAN/MAN Networks", *IEEE Journal on Selected Areas in Communications*, pp. 1172-1182, Oct. 1993.

[7] Peter Salamon, Paolo Sibani, Richard Frost, *Facts, Conjectures, and Improvements for Simulated Annealing*, Philadelphia: SIAM (Society for Industrial and Applied Mathematics), 2002.

[8] Albert-Laszlo Barabasi, and Reka Albert," Emergence of Scaling in Random Networks" Science , 286, (1999) 509-512.

[9] D.S. Callaway, M.E.J. Newman, S.H. Strogatz, and D.J. Watts" Network Robustness and Fragility: Percolation on Random Graphs", Physical Review Letters, 85, (2000), 5468-5471.

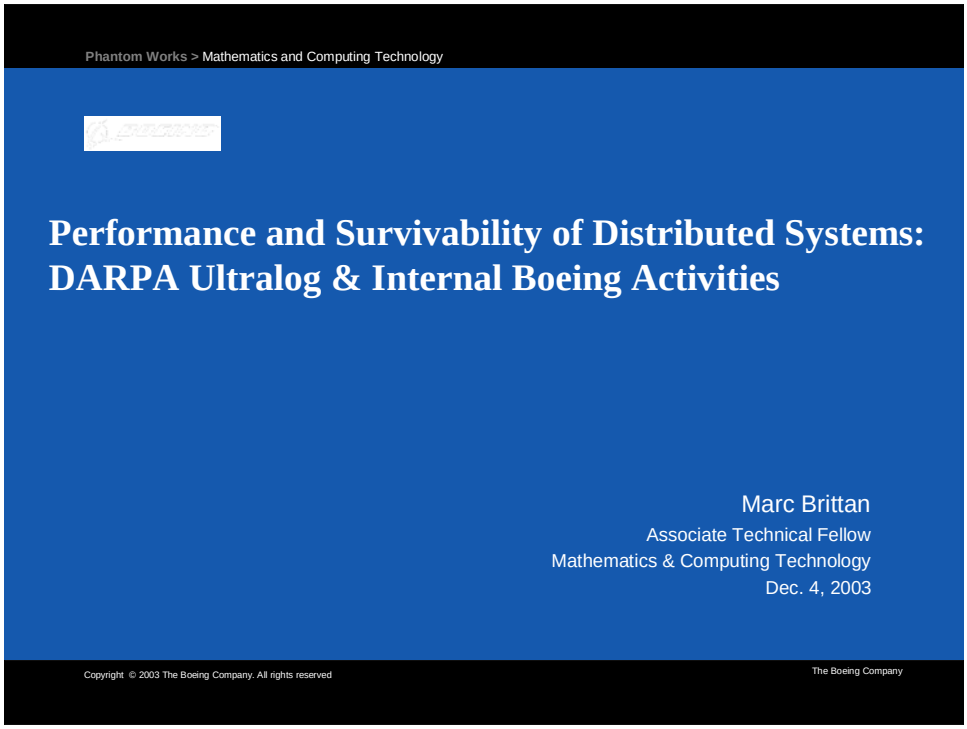
[10] Reuven Cohen, Keren Erez, Daniel ben-Avraham, and Shlomo Havlin, "Resilience of the Internet to Random Breakdowns" Physical Review Letters, 85, (2000), 4626.

[11] S. Lin and B. Kernighan, "An effective heuristic algorithm for the traveling salesman problem". *Operations Research*, 21:498-516, 1973.

12 Appendices: HPC 2004 Conference PowerPoint Presentation

This appendix includes by reference the PowerPoint presentation slides that were used during my talk at the “High Performanc Computing Conference” in Cetraro Italy. The talk was given on June 2, 2004, and summarized our Load Balancing efforts and suggested enhancements to improve performance of distributed agent systems.

The PowerPoint presentation is included in the file titled “hpc2004.ppt” on the documentation disc, and is summarized below in graphic (non-PowerPoint) format.



Autonomous Systems

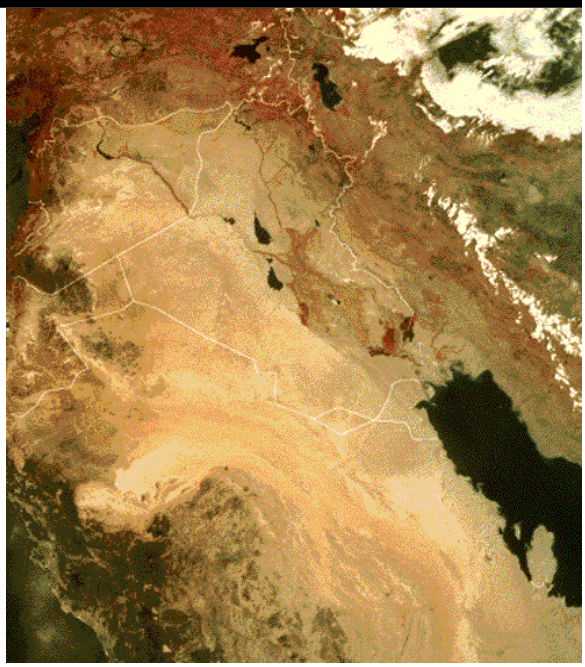
RED Link = Optimal State

BLUE Link = Failover State

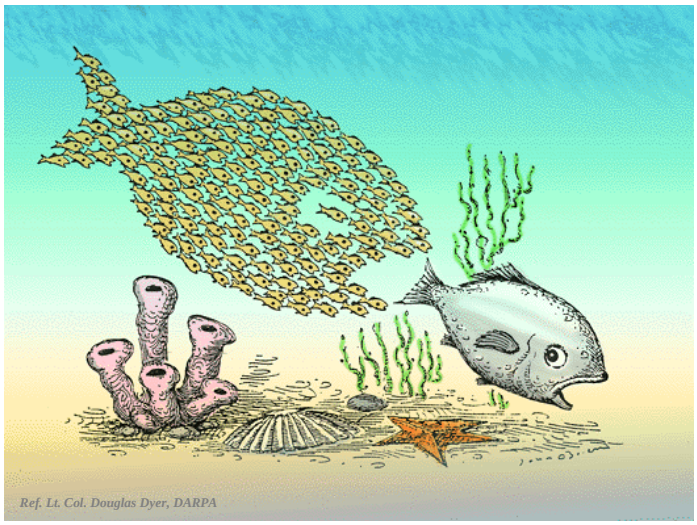
BLACK Link = Dead or Unusable Link

YELLOW Circle = Point of Attack

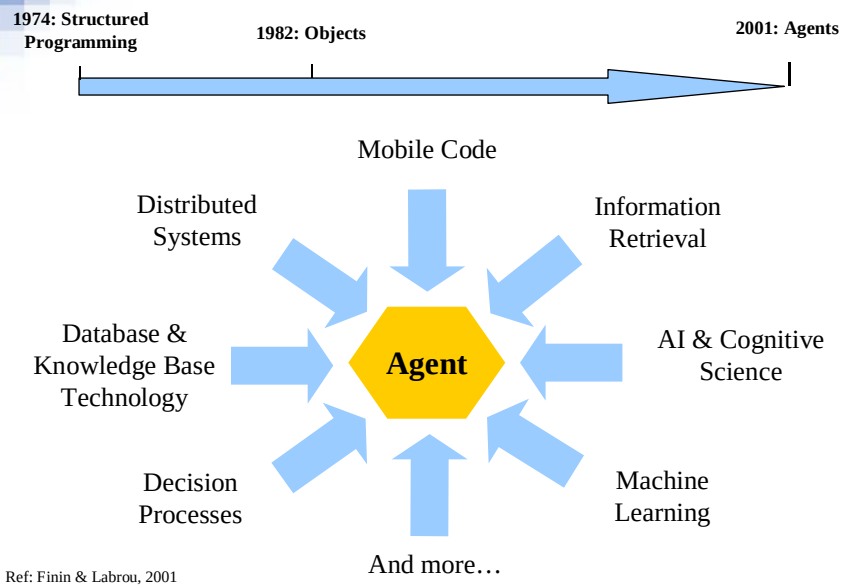
Line Thickness = Frequency of occurrence of that link in the set of all failover plans.



Agents: The Power of the Collective



Agents: A System-building Paradigm



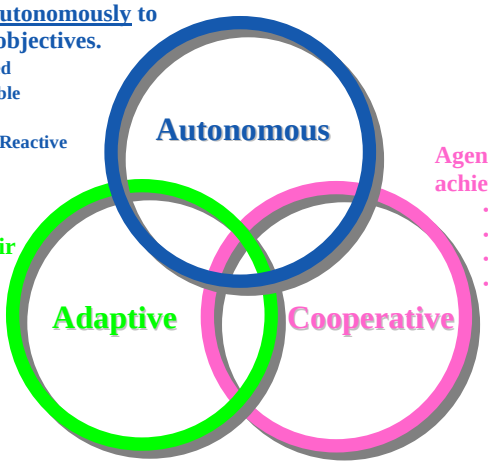
Ref: Finin & Labrou, 2001

Key Agent Characteristics

Agents act autonomously to accomplish objectives.
- Goal-Directed
- Knowledgeable
- Persistent
- Proactive & Reactive

Agents adapt to their environment.
- Dynamic Interaction
- Alternate Methods
- Machine Learning

Agents cooperate to achieve common goals.
- Communication Protocols
- Knowledge-Sharing
- Coordination Strategies
- Negotiation Protocols



Note: Agents can be either static or mobile, depending on bandwidth requirements, data vs. program size, communication latency, and network stability

Derived from: H. Nwana, Software Agents: An Overview

- **Future systems must plan for attacks, and component failure**
  - **Some distributed tasks take hours/days to complete**
    - *Gradual degradation is vital in these systems! Need reliable backup and failover plans that also meet performance requirements!*
  - **Current systems subject to DDOS, Kinetic Attack, Spoofed IP's etc. complicate defensive measures**
    - *Need system of mobile jobs, or agents, to create a moving target for attackers, and create a natural defense to targeted attacks.*
  - **Defenses should plan for coordinated attacks on the system performance solution space**
    - *Attackers are getting smarter, so our systems must get smarter...*

DARPA Ultralog Goal: ***"Operate with up to 45% information infrastructure loss in a very chaotic environment with not more than 20% capabilities degradation and not more than 30% performance degradation"***

- **DARPA Ultralog is a robust and secure system of mobile agents designed to enhance distributed system survivability, performance, and security.**
  - Agent mobility creates flexible workloads that effectively complement flexible routing, enhancing system survivability
  - ***Ultralog has been a DARPA program for 3 years.***
    - Highly flexible architecture allows rapid system redesign before, during, and after attack or failure.
    - Manages remote backups of benchmarked states of system
    - Ultralog architecture has potential for military and commercial applications far beyond its current use in Military Logistics.
    - ***Boeing*** *wrote the "solver" for the near real-time management of System Performance and Risk.*

Boeing's prior work in automating Distributed System Design put it in a good position to design software to manage performance and risk in Ultralog.

- Prior to Ultralog, Boeing developed "Ex Nihilo" tooling for automated design of distributed systems.
  - Tool addressed common problems at core of distributed design and task scheduling
    - Based on Combinatorial Optimization in a Faulty Queuing Network
    - Analyzes multiple parallel threads for response time, risk, cost, etc.
    - Accounts for Random Processes via analytic M/M/c queuing theory
  - Based on fusion of earlier Boeing work in Distributed System Performance Modeling and work in Mission Planning, Decision Design, and Command and Control (BMC4I)
- State Management problem of assigning jobs/agents to servers has many hard subproblems
  - State is defined as agent/node assignments and routings.
  - Numerous NP-Complete constraint problems, like CPU, Memory, Bandwidth constraints.
  - #P-Complete reliability problems, like classic source-target network reliability mixed with alternate agent/node assignments.
  - ***Many problems in classic distributed system design must now be solved in near real time***

Physical View of System (Physical Topology)

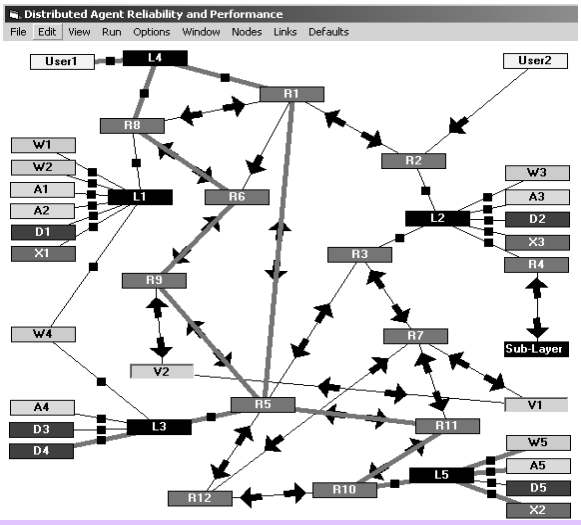


Figure 1: The current optimal state is highlighted with thick black lines. Jobs/Agents are allowed to run on any of the eligible server types (e.g., Web jobs optionally run on servers W1-W5).

Managing Server Health and Constraints

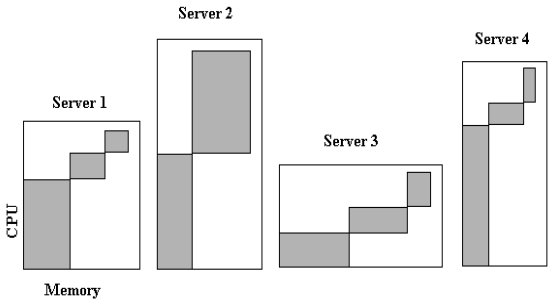
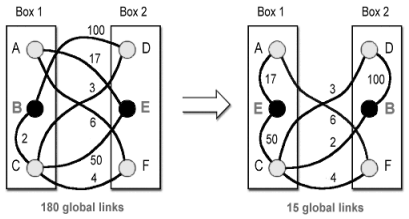


Figure 2: Assign agents/jobs to servers to ensure system health. In this example 11 agents (smaller solid blocks) are assigned to 4 servers. CPU is the vertical dimension, and Memory is the horizontal dimension.

- Healthy system means each agent gets its required CPU and Memory
  - Minimize context switching and queuing at CPU due to overloaded processors
  - Minimize page faults due to inadequate memory (use of virtual memory)

Managing System Traffic

Minimize Remote Messaging (Graph Partitioning) :



Results of exchanging agents B and E:
- Remote messaging rates decrease
- Reduced network delays !

Figure 3: The two servers on the left have a high rate of communication between servers. To improve performance, we exchange agents B and E, which produces a system with lower inter-host messaging.

- Graph Partitioning is a classic technique in distributed system design
  - Minimizing remote messaging is a fast approximate design method
  - Slow WAN/LAN delays are typical bottleneck in response time
    - True response time minimization requires thread tracking and optimization
      - *Thread tracking and optimization is performed in Boeing Ex Nihilo tool, but not implemented in Ultralog*

Managing Risk

Currently manage risk at two levels, in two passes through Ultralog state optimizer:

- First pass: Whole system risk due to all sources of failure (maximize "up-time" for the system)
  - A state failure occurs when at least one physical component required to run the state fails
  - Maximize Psuccess(system), where Psuccess = 1 - Pfail
    - **Psuccess(system) = $\Pi_{Component \in State}$ (Psuccess(component))**

- Second Pass: Agent-weighted risk, minimize number of agents on risky servers
  - Example: All agent weights equal, minimize the maximum expected damage from a single point attack
    - **Minimize { Max(inod∈system) [Pfail(inod) * NumAgents(inod)] }**

    Pfail(inod) = Probability of failure for node number inod
    NumAgents(inod) = Number of Agents on node number inod

Solver Details

Combined problem of health, risk, and traffic management is classically intractable, and requires special techniques to manage the system.

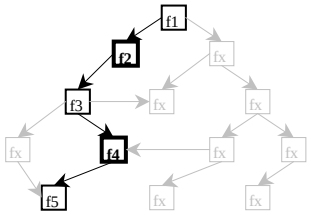
- Solver must find solution within constraints that ALSO minimizes risk and manages traffic
  - Overall problem complexity is at least NP-Hard
    - Although NP-Hard, problem is easy in practice in Resource-rich environments
      - Simple Greedy searches work well
    - Problem is Difficult in resource-marginal, or resource-starved situations (Battle Applications *MUST* prepare for all environments)
      - Full NP-Hardness is felt in tight resource situations
      - Special techniques required to solve problem

Solver Details (cont)

- Solver based on Simulated Annealing, with other techniques as needed
  - Relies on theory of Inhomogeneous Markov Chains
    - Temperature is chain parameter, which is measure of randomness of search
  - Solver hops from state to state, gradually lowering temperature
    - $State_1(temp_1=hot) ==> State_2(temp_2) ... ==> State_k(temp_k=cold)$
    - Hot Temperature ==> Pure Random Search
    - Cold Temperature ==> Gradient Descent-like
  - Simulated Annealing is extremely Robust compared to other global optimization techniques
  - With special operator selection and tuning can be competitive with other fast algorithms
    - No Free Lunch Theorem: No single best optimization technique

Neighbourhood Structure and Local Search

- Annealing algorithms require careful attention to topological structure of solution space (like all local search algorithms)
  - Neighbourhood structure is problem dependent
    - Some basic state transitions common to all problems
      - Move agent/job from one random server to another random server
      - Augment basic state moves with problem dependent moves
      - Risk related problems:
        - ✓ Move agents/jobs from high risk servers to low risk servers
      - Response time and thread tracking problems:
        - ✓ Move agents/jobs between random threads

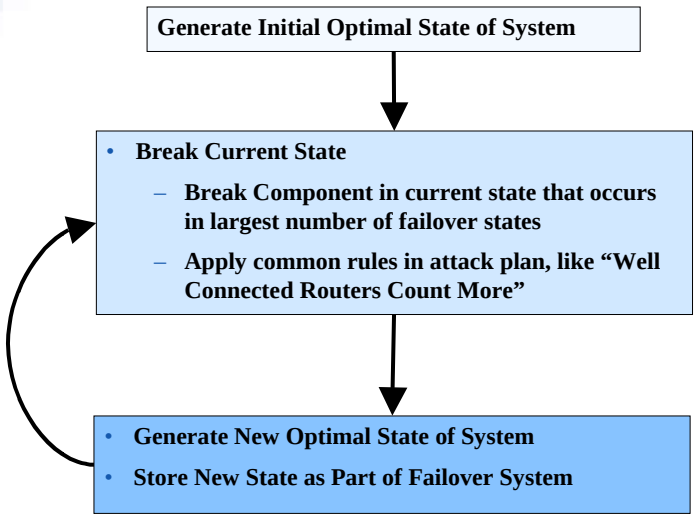


Survivability and Failover Planning

To kill a distributed system, we need to kill both the current mode of operation, and destroy the ability of the system to self-heal, or rebuild itself into a new working system, or subsystem.

- Since solver has best knowledge of "System Performance Solution Space", have solver attack itself
  - Generate an iterative sequence of Optimal States and Optimal Attacks on States
    - Preceding discussion describes steps in building Optimal State
    - Attack is based on a broad and deep strategic attack, attempting to destroy components that occur in a large number of failover states of the system
    - Compare to tactical attack that destroys only the current operating state

Solver Self-Attack Algorithm



Destroying the Ability of a System to Self-Heal

Define Logical Topology $\mathfrak{I}$ found by solver:

$\mathfrak{I}$ = Set of all states of system that meet hard constraints

P($\mathfrak{I}$, *objectid*) = Probability that random selection from $\mathfrak{I}$ contains ***objectid***

where ***objectid*** is a Link, Router, Hub, Server, or other element of the physical topology

- **Consider 3 subcases for the set of Logical Topologies $\mathfrak{I}$:**
  - $\mathfrak{I}_{\#}$ = Set of ALL possible failover states or logical topologies
    - # symbol used to highlight association with #P-Complete Network Reliability problem
    - Set of all states is too large to be useful in real-time
  - $\mathfrak{I}_{failover}$ = Set of topologies associated with stored failover plans
    - Cannot be computed in real time, but can be used in real time (compute in background)
  - $\mathfrak{I}_{anneal}$ = finite set of stored failover plans
    - Less accurate, but can be computed in real time
    - **P($\mathfrak{I}_{anneal}$, *objectid*)** found by exponential damping of solutions found by solver as temperature is lowered

Example of Solver Self-Attack Using $\mathfrak{I}_{anneal}$:

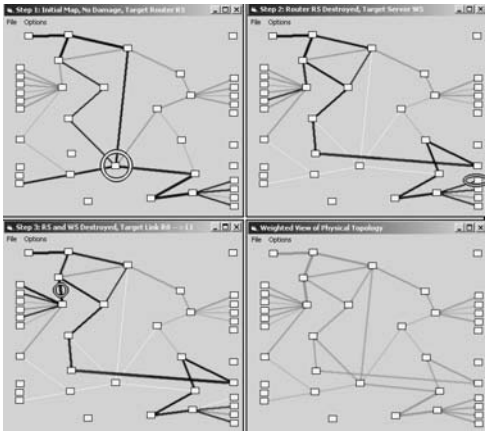


Figure 4: Four views of the solution space. The first map is for a fully operational system, followed by two maps generated after attack. Circles show point of next attack. Optimal solution is in black, with backup solutions in gray. White lines are for unused links that were used in previous steps. Fourth map represents a weighted summary of the solution space for distributed design, with thicker lines used for links in more failover plans.

Conclusions:

- **DARPA Ultralog is an extremely robust and survivable distributed system based on mobile agents**
  - Marriage of highly flexible software architecture and highly flexible hardware architecture
  - Boeing wrote the solver for management of Performance and Risk in Ultralog
- **Boeing system design tool is an enhanced version of Ultralog solver**
  - Includes queuing theory and multi-threaded tracking
  - Addresses a number of NP-Complete and #P-Complete problems in distributed system design
  - Boeing tool had its roots in BMC4I modeling and optimization

Future Work:

- Natural extension of current Boeing tool: Return to its roots, and rework for Command and Control Decision Design
- Topology Management for advanced networking
- Incorporate into mission planning work to plan effective air and ground based networks
  - Incorporate techniques into other Adaptive and Autonomous Networking projects

13 Appendices: Distributed Design White Paper

This appendix includes the white paper that was presented to the group in 2002. The paper is a collection of a large number of email messages that were posted to the email lists during 2002 and reformatted into a document.

Intractability in Distributed Systems Design: The Ex Nihilo Solver

By Marc Brittan

Mathematics and Computing Technology,

Boeing

September 1, 2002

CONTENTS:

Suggested Future Topics:

*) Non-Markov Processes (e.g., Chaotic Performance)

*) Time Slicing, Queuing, and Processor Affinity (Context Switching)

*) Transient Analysis

*) Performance Load Testing for Agent Resource Requirements

*) Task Creation and Deletion

*) Game Theory

*) Fault Detection in Networks

Preface:

This white paper is a collection of a number of posts made to the Ultralog Scalability mailing list by the present author and developer of the Ex Nihilo distributed design tool. It also contains some updates to the posts based on things learned during the project. As a result of this mailing list format, the document still has an informal writing style that is appropriate for a mailing lists, although inappropriate for a formal document. Although we have made attempts to improve the formality of the posts, to create a somewhat more formal document, it is still in an informal format, intended as a working document on ideas in Ultralog. I have inserted a few comments into the commentary to point out changes that have been made since the initial writing.

Forward

I really enjoyed meeting everybody at the workshop in Virginia. According to mailing list canon, it is the practice to introduce yourself when making a first post to a list. I am Marc Brittan, from Boeing, Mathematics & Computing Technology, where we developed our "ex nihilo" tool, for designing large distributed systems. Our Ultralog proposal was to modify this tool to operate in a distributed agent environment. I am quite confident that our existing system design tool can be modified to operate on the distributed agent reliability and performance problem. For problems up to a certain level of difficulty, the tool should be able to make operational suggestions in real time. It could also be used "offline", to design a survivable system, with planned backups (failover modes), and planned performance and reliability. Our current design tool also has a "cost model". The costs from the cost model are used as input to the objective function (measure of goodness), which includes fixed and monthly costs used to find cost effective designs and modes of operation. Effective cost modeling is extremely important in the commercial environment, for which the tool was developed. In today's environment of tight budgets, I am confident that cost modeling will also be of interest to the Government.

Introduction

The tool was initially designed to handle some of the "intractable design decisions" that come up in designing a multi-tiered client server system. The target application was to design multi-tiered client server systems, like classic 3 tier systems, to minimize Response Time, Probability of Failure, Fixed Cost, and Monthly Cost. Although we could add more components of "quality of design", these four areas capture most topics of interest to system users (Response Time, Failure), and of interest to "those who pay the bills". Obviously we will be adding many more measures of goodness during the Ultralog project, and we need to translate those measures into numbers for routing, scheduling, assigning, backup planning, design, and operations.

There are a large number of intractable subproblems in both classic multi-tier design problems, and more general distributed design problems. Even some common statements of "satisficing", as discussed during our meeting in February, can be shown to be NP-Complete. I will discuss the satisficing parts of the problem below, but I bring them up now, since we knew we would need to solve these types of intractable problems with our distributed design tool. The need to solve these hard problems in distributed systems design, including intractable "satisficing problems" as discussed at our Feb. meeting, figured prominently in our software design decisions for our new distributed design tool, "Ex Nihilo". The phrase ex nihilo is Latin, and is usually translated as "out of the vacuum". We wanted to build a process/tool that would allow the user (a designer of distributed systems) to design systems with near optimal measures of response time, failure, and cost. The tool should take the user from a "clean sheet of paper(vacuum)", to a finished design. Building the tool would involve some of the most complex problems in networking, and some of the most important (and difficult) problems in mathematics.

We designed our "design tool", ex nihilo, to model general topological designs, with complex connectivity and routing. Since the multi-tier design problem was intractable, it would require special techniques for some of the intractable design decisions. Furthermore, these techniques (graph theory, queuing network theory, simulated annealing...) are in some sense independent of the underlying topology. What this implied, is that constraining the tool to classic multi-tier architectures would not really buy us anything, since the multi-tier design problem was already intractable. Based on this, we decided to build a design tool that would solve general distributed design problems, for general architectures.

While we did target a broad class of architectures, there are some required modifications to both the solver and GUI of ex nihilo that are required to address the distributed agent problem. These modifications are outlined in our Proposal, and our Statement of Work. The most significant of these is the main task scheduled for this year. Although the solver and GUI treat a general topological design, they do need to be modified so that each "host" in the system can be the "center of the universe" for calculating response time calculations. The model currently makes response time calculations based on a single class of user, so in this sense, it still has a tiered architecture centered at the single user class, even though the solver and GUI handle general topological architectures once a request (program thread) has been initiated by the user.

For the distributed agent problem, where tasks may be moving around, and where initial tasks (top of thread) may occur anywhere on the system (any host), we clearly need to be able to measure response times from anywhere on the system. This is a major architectural rework of ex nihilo, touching all of the major data structures and methods in the program, and is a key task in future adaptation of ex nihilo to the distributed agent problem. Mathematically, this is an insignificant increase in program complexity, although programmatically it is a lot of work for the programmers. It is also the backbone of all future enhancements to response time modeling. The "Multi-User class extensions" task, which tracks response time from multiple points in the system, is now scheduled for 2003-2004.

So now that we are done with introductions, I would like to discuss some technical issues brought up in Virginia which require a more detailed response than that given during our short time in the workshop. Since there are a large number of technical issues that came up at the last workshop, I will break them up into a series of messages. I will try to be careful in explaining some terms that come up in the discussion, since we all have different backgrounds... and I ask your patience while I come up to speed in some of the Ultralog terminology, like Enclaves, Nodes, etc.

1. Routing, Satisficing, and Greedy Search

Router Programming:

I would like start this series of messages by expanding on some of my statements about routing during our recent Load Balancing meeting in February. During the meeting, I stated that up until that point in the meeting, we had only discussed node and host based inputs and outputs to ex nihilo. I mentioned that ex nihilo makes recommendations for router programming (router tables), and that this information could be used in the design or operation of the system. A member of the load balancing team asked how the program was using this information to find solutions, and I responded that the program starts with a "physical topology", and then determines routes based on the current search goals (mix of goals and constraints in response time, failure, fixed cost, and monthly cost). Since the overall network has many possible routes for large systems in the physical topology, the routers choose "optimal" routes for the network messages (packets). These router selections determine a "logical topology", which very loosely speaking, is a subset of the physical topology. The person asking the question then mentioned that for our purposes, the system starts with a logical topology, so the router settings are given. This is the point where I would like to extend the discussion, and focus on some of the implications.

"Load Balancing" of Hosts, Links, and Overall System:

While classic router programming is a relatively simple problem, the problem of designing a set of router tables that produce a system that operates at reduced cost, minimized failure probability, and reduced response time, is classically intractable, and hence appropriate for the techniques used by ex nihilo.

For our first demo this year, the fact that we are starting with a logical topology, as opposed to a physical topology, is very good news. The router search is performed at every step of the search, so eliminating the routing search will result in considerable speedup of the solver. The capability to design better routing matrices is already in the ex nihilo solver, but is simply not needed for this year's demo's.

Complexity of Routing and Other Problems:

Let's continue with our discussion of routing issues in a bit more detail. The classic "shortest path" routing algorithm is a fairly basic task, and is solvable in "polynomial time" on computer (Class P), which is math-speak for saying that the time to find the optimal path between two points can be bounded by a polynomial based in the size of the network. For example, some problems grow quadratically in difficulty (x**2), so increasing the size of a problem by a factor of 10 (e.g., increasing from 10 nodes to 100 nodes), might increase the running time on computer by a factor of 100. Quadratic growth is not a big problem on computers. In computational complexity, anything less than "exponential growth" is considered "good".

The problems we are currently faced with, at the heart of Ultralog, are the problems that appear to be exponentially difficult (exp(x) versus x**2), or worse. These types of "exponentially difficult problems" are usually referred to as "intractable". For small problems, we can find the solution, by exhaustive search, in fractions of a second on a PC. For large problems, finding the EXACT solution (optimal solution in optimization, or correct yes/no decision in decision problems), can take longer than the current lifetime of the universe, on our fastest computer. So the state of the art is to use techniques like simulated annealing, genetic algorithms, rules, and anything else around that works, to find the best solution we can, in the allotted amount of time. For problems like this, we search for good solutions in a reasonable amount of time, compared to finding optimal solutions in an unreasonable amount of time.

There is a special class of "decision problem" (Class NP-Complete), for which there are no known polynomial time algorithms. These decision problems are "Yes/No" decision problems. It is important to draw a distinction between decision problems and optimization problems. The most famous NP-Complete problem, the Traveling Salesperson Problem (TSP), is a decision problem that is usually stated (roughly) as "Is there a tour of the N Cities that is less than X miles?". This is a Yes-No decision for a computer... good for binary things in particular ;)

The "optimization version" of the Traveling Salesperson Problem is "Find the shortest tour of the N cities." This problem is not stated as a Yes-No problem, so the phrase "NP-Complete", does not apply in the literal sense. The decision/optimization problems may be related, but there is a distinction. In this case, the TSP-decision problem is "NP-Complete", and the TSP-Optimization problem is referred to as "NP-Hard". It is clear, that if we have a "fast" algorithm (polynomial time on computer) for the TSP-Optimization problem, then we can also solve the TSP-decision in polynomial time. All we would need to do is find the optimal tour, and answer the Yes-No decision problem "Is the tour less than X?". So the NP-Hard problem appears to be as difficult, and "practically" more difficult, than the NP-Complete problem, although both are referred to as intractable.

Decision, Optimization, and "Satisficing":

I am highlighting this difference between NP-Complete and NP-Hard, because there was another question during the meeting on whether "Satisficing" might be more effective than "Optimizing". This is similar to the distinction between NP-Complete and NP-Hard. There are a number of typical "satisficing" constraints on Ultralog Performance and Failure that have subproblems that are NP-Complete (and therefore believed to scale exponentially). The optimization versions of these same problems are NP-Hard. A common example of a "satisficing statement", is to ask that the system complete a specified task, distributed throughout the network, in a fixed amount of time (e.g., satisfactory response time is 5 secs.). The optimization version of this problem searches for the minimum response time for the task. So while "satisficing" does appear to reduce the complexity, the underlying problem itself remains NP-Complete, even in the satisficing problem. It is in this sense that "satisficing is easier than optimizing", namely, the sense that "NP-Complete is easier than NP-Hard", although both are classically intractable.

It is also important to realize that the actual solution techniques for satisficing are frequently identical to optimization. For example, to find a "satisfactory solution" for response times, we typically run the optimization until a solution is found in which run time meets the "satisficing constraints". Once a satisfactory solution is found, we can then terminate the optimizer - the solution is "good enough".

Ex Nihilo Solver:

Ex nihilo allows a mix of optimization goals and constraints (min, max), so the suggestion of relaxing things a bit, and "satisficing" as suggested in our February meeting, is good news for solver run time. The "satisficing capability", in the sense of constraints set on response time, failure probability, fixed cost, and monthly cost, is already present in ex nihilo. It may be desirable to add other measures of goodness, or penalty, to the model. The ability to change things (like constraints and objectives) in the simulated annealing algorithm is a very positive feature of this technique, and is one of the reasons this technique is frequently preferred over competing techniques like integer linear programming, genetic algorithms, and others. It is also fairly easy to change this part of the solver, to add additional constraints, goals, and rules, such as "Probability of Security Risk" and other things of interest to the user and operator community.

The ability to easily modify the objective function, and its representation, is a major strength of the simulated annealing technique. There are a number of techniques in optimization that are referred to as "brittle". These techniques may work great in a limited domain, but may be worthless when the objective is modified (e.g., squared), or when the constraints are modified. The simulated annealing technique is one of the most "robust" algorithms in discrete optimization. Although the heart of the ex nihilo solver is simulated annealing, the solver is intended to use techniques that have a genetic flavor for selecting moves through the "solution space". So our actual solver is a blend of techniques in annealing, genetics, and classic network mathematics, taking the "good parts" of each of these techniques.

Greedy Search:

Another comment made during the Load Balancing breakout session suggested that we should consider Greedy Search and other techniques instead of global optimization by simulated annealing. While greedy search can be effective at finding approximate solutions in a short amount of time, the technique is notorious for occasionally getting trapped in local minima, and missing the global solution, or even approximate global solution.

Although we planned a simulated annealing search engine for the core of ex nihilo, our very first code for ex nihilo, and our first operational prototype, used a greedy search technique. Due to the limited amount of time in our first workshop, I did not demonstrate the Greedy Search part of the tool. This greedy search can be used for a "starting guess" at the initial annealing solution. A good starting point for any optimization problem can result in considerable speedup of the solution process, so a fast greedy search has been in the solver for over two years now. If the starting guess is good enough, we can skip the annealing phase of the solution.

The modification of the greedy search for distributed agents, and the use enhanced operators for moving about the "design space", is already in our schedule. The task in our Statement of Work (SOW) labeled "Modifications to Improve Convergence" is directly associated with incorporating these techniques to speed up the solution process. The task labeled "Analytic Seed of Solver" is directed at the use of other techniques to solve sub-problems of the overall agent reliability and performance problem. Another task, labeled "Investigate Analytic Estimates in Support of Non-Markov Processes" is also meant to address this aspect of using other solution techniques. The modeling of non-Markov processes is a very detailed matter, and will be addressed under a separate heading.

2. Thrashing and Satisficing

I would like to comment on some specific issues that came up during our meeting in February. In the last section, I discussed Routing, Satisficing, and Greedy Search. In this section, I would like to talk about the problem of thrashing, or needless changes in state. The thrashing issue will be related to the Satisficing comments in the last section, and we will discuss how some satisficing measures might be implemented in ex nihilo to make "reasonable changes in state".

Thrashing, Satisficing, and Minimizing State Changes of the System:

During our Feb. 2002 workshop, the issue of "thrashing" came up within the context of state changes of the system. It was suggested by a member of the load balancing team that satisficing, as discussed in the previous section, might be effective at minimizing thrashing, which for our purposes will be defined as excessive changes in the system, with minimal difference or improvement in the states being "thrashed".

It should be clear that some measure of satisficing will be necessary for the successful operation of the system. It is also important to draw a distinction between suggestions made by ex nihilo, and actual operation of the system. The operation of the system, and decisions on whether a change in state is required, are perhaps best handled by an external "oracle", which would make decisions on state based on inputs from ex nihilo, inputs from other systems agents, and perhaps inputs from a human in the loop.

I would like to discuss how satisficing is currently treated in ex nihilo. During our demo at the workshop, we didn't have time to discuss details of the objective function, which is our measure of goodness being optimized by ex nihilo. When I first designed the tool, it became obvious that there were a couple of key operational constraints that would impact the system designs found by the tool. We will illustrate these constraints by an example, and focus on response time, although the same techniques have been implemented for the other metrics like probability of failure, monthly cost, and fixed cost.

Maximum Constraints: (Hard or Soft)

Consider a user (agent) that has a maximum acceptable response time for online tasks. For example, a user making real-time queries to a distributed database system might say that a response time of 20 seconds is "satisfactory", with any response time greater than 20 seconds defined as "unsatisfactory" (add shades of gray as appropriate).

This maximum constraint is implemented in the ex nihilo tool as a hard constraint. When the solver finds such a state, it rejects it as a viable solution by setting a high value for the objective function (we minimize the objective function). This is our first level of satisficing in the model. Any solution with a response time less than the maximum constraint is defined as satisfactory. An obvious extension, which would also add to solver stability, would be to change this hard user constraint into a soft constraint. For example, we might set 20 seconds as a satisfactory constraint, with a steep decline in goodness for values greater than 20 seconds, as opposed to "a brick wall" at 20 seconds. This could be implemented by having an exponential or other damping in the objective function for response times greater than 20 seconds. As mentioned previously, the ability to readily change the shape of the objective function is a strength of the simulated annealing technique.

It is worth mentioning that the actual user-defined objective function may be modified by the solver in order to better explore the design space. For example, we might "relax" some of the constraints during the early part of the solution process, effectively flattening out the objective function. This relaxation allows the solver to more easily move across some of the "hills" that separate the "valleys" of our objective function. As the solution progresses, the relaxed objective/constraint is allowed to return to its original user specified value, so the final solution will meet the user defined constraints. Note that the solver may violate some of the constraints during the solution process. This is a fairly classic approach in optimization, which allows the solver to explore the whole space, and avoid getting trapped in local minima.

This maximum constraint in ex nihilo is just one aspect of satisficing that is already in the program. Any of the solutions found by ex nihilo that fall below the maximum constraints can be defined as "satisfactory". If the current state of the system is satisfactory, and there is no reason to change state (other than external intelligence), then there is no need to sample from the set of state recommendations from ex nihilo. If there is some reason to change state, then the agent(s) managing Ultralog performance and reliability could consult the recommendations from ex nihilo in making state change decisions.

Minimum Constraint: "Equally Good Solutions"

In addition to the hard maximum constraint in response time (or softened in a future tool), there is a minimum constraint in the tool that can be defined by the user. This minimum constraint in ex nihilo could be used in Ultralog to further minimize thrashing of operational states of the system.

There is a major difference in interpretation between the minimum and maximum constraints in ex nihilo. While the maximum constraint is the boundary between satisfactory and unsatisfactory solutions, the minimum constraint is used to define a set of "equally good" solutions. Since the original target audience of the tool was the set of real human users, online, in a distributed environment, then this gets into issues of human factors in the design of distributed systems for real-time use.

Returning to our example, a user (or management team) might choose 20 seconds as the maximum acceptable response time for a computer task. This value might be based on the assumption that during this response time, the user is going to be sitting in front of a terminal, waiting for a response... and getting no useful work done while waiting (lost productivity). It is difficult for humans to time-slice their workload at the 20 second level. To borrow a term from the computer performance industry, their is a "context switching penalty" each time a user switches tasks. For a human user, this response time is wasted time, and if the time exceeds this 20 seconds, it is likely that the users attention will drift away, to that beach in Hawaii... The maximum response time limits are usually much longer for batch jobs than online transactions, although the mathematics of the problem is the same.

At the other extreme, a human user cannot take advantage of extremely short response times, since the human response time is longer than the system response time. For example, our user might find response times of .1 seconds and .01 seconds as "equally good", since they could not take useful advantage of the difference in delay times.

The set of "equally good solutions" is frequently said to comprise a set of "degenerate states". For reasons associated with convergence, the ex nihilo solver will hop around randomly among degenerate states when they are discovered.

If the "oracle" described earlier knew of the current operational state of the system, it could use samples from ex nihilo of both the degenerate states (nearby or "equally good" states) and the improved states. These samples could be generated on an ongoing basis, with ex nihilo running as a daemon, and used by the oracle to make decisions about operational changes of state.

"Shape" of the Objective Function:

The current objective function is flat at the bottom of the response time curve, so the curve is flat from zero up to the minimum response time constraint, and is numerically equal to the value of the objective function at the user defined minimum constraint. The objective function then has a user definable fixed slope, until it reaches the maximum constraint, at which point it hits the "brick wall", and becomes infinite (big). This captures the major response time design issues of interest for a distributed system, namely, the concept of a maximum response time, with increasing value of "goodness" for shorter response times, until some minimum response time limit is reached, at which point further improvements are a waste of money (so no supercomputers for the spreadsheet users).

All of the current major input goals and constraints to ex nihilo (response time, failure, fixed cost, monthly cost), have this basic overall shape for there contributions to the overall objective function. The overall objective function in the current tool is a weighted sum of the values of the individual functions for the four major components.

Conclusion:

To some degree, we already have the ability to handle some measures of satisficing in making projections for recommended states of operation. As mentioned previously, the solver is fairly easy to modify for most types of objective functions and constraint sets. The shape of the objective function, based on inputs for response time and other goals/constraints can be easily modified to other desired forms.

The next section is going to be about "Distributed Use of Ex Nihilo", where the goal is to operate a set of mirrored ex nihilo solvers, needed to avoid having the optimizer be a single point of failure.

3.0 Distributed Use of Ex Nihilo

Our next topic involves a question brought up in the February 2002. workshop on the distributed use of ex nihilo. To minimize ex nihilo as a single point of failure, it was suggested that an optimizer like ex nihilo should be operated in a parallel mode, with mirrored copies of the optimizer running on a set of nodes in the Ultralog part of the net.

In the last couple of sections, we discussed how the tool generates routes and task assignments to optimize (or satisfy) goals and constraints on response time, failure, fixed cost, and monthly cost. We also discussed how issues like thrashing (needless state change) could possibly be addressed using some of the key constraints (min, max) of the objective function inputs, along with adjustments to the overall shape of the objective function. All of this previous discussion was in the context of a single instance of the optimizer, located at a single point on the Ultralog net. We will now look at some of the problems that arise in the distributed, mirrored operation of the optimizer.

Distributed Use of Ex Nihilo:

During the load balancing breakout session, we discussed operating ex nihilo in a distributed mode. Having the optimizer run on a single host would make it a single source of information for reliability and performance, which would make it a single point of failure. To improve survivability, the agent reliability and performance tooling should be designed and mirrored to operate throughout the network, as with the rest of Ultralog.

As long as the optimizer has the same data input, it will find the same answers (within reason for a random search), regardless of where it is located on the net. So having mirrored solvers on a diverse set of servers throughout the net is not a problem, other than the usual problems with server mirroring...

Server Mirroring for Ex Nihilo:

In the first section, we highlighted some of the NP-Complete and NP-Hard problems in the distributed computing design problem. In this problem, we assign tasks to hosts on a network, designed to meet performance, reliability, and cost goals and constraints. The problem of server mirroring is a famous hard problem, and has subproblems that are known to be NP-Complete. The server mirroring problem is going to be one of the keys to distributed computing in the future.

In the server mirroring problem, we decide where to place copies of a servers throughout the net, such that a set of users are within some specified measures of cost and distance in a network. Following is a very formal statement of a typical server mirroring problem. It is listed as a problem in one of the best known (and highly recommended) books on Computational Complexity. We have left the problem in its original graph theory notation to give the reader a flavor of a formal problem definition. In graph theory, we define sets of nodes and edges connecting the nodes, with varying weights. Graph theory problems occur in many places in distributed design.

The Graph G described below is made of "vertices" v, and "edges" e. The vertices in V in the graph are data elements or jobs on servers, while the edges in E are the data links connecting the servers.

INSTANCE: Graph G = (V,E), for each $v \in V$ a usage $u(v) \in Z^+$ and a storage cost $s(v) \in Z^+$, and a positive integer K.

QUESTION: Is there a subset $V' \subseteq V$ such that, if for each $v \in V$ we let d(v) denote the number of edges in the shortest path in G from v to a member of V', we have

$$\sum_{v \in V'} s(v) \quad + \quad \sum_{v \in V} d(v) * u(v) \quad \leq \quad K \quad ?$$

COMMENT: NP-Complete in the strong sense, even if all $v \in V$ have the same value of u(v) and the same value of s(v). Note that our distance metric, in this case, is phrased as number of hops in a network. [Garey and Johnson, "Computers and Intractability: A Guide the Theory of NP-Completeness"

This formal problem is a famous problem at the foundation of distributed design. A rough interpretation, is that in the above equation, we are adding cost of storage per host (however measured or interpreted) for each data or processing element on the system, to a distance term that sums over all users. The users are going to be accessing the processing/data on the servers at varying rates, and they will want to access the closest server in the set of mirrored servers. In that distance sum, it multiplies the distance from that user to the user's closest mirror server, times the usage rate of that user.

Ok, not an easy read, but the idea is that the famous server mirroring problem captures the idea of server cost plus cost of traffic from users to server mirrors. We want the distance (measured in network hops in this example) between the users and the mirrored servers to be low. So the function d(v) is the shortest path, for each user v, from that user to a server. The cost of storage on servers for data elements (or tasks) v, is expressed as s(v) in the above equation. The usage function u(v), is a measure of traffic or usage, from each user v to a server. The cost can be based on a variety of factors, like real cost, or interpreted cost issues like risk and other measures.

This statement of server mirroring is a good first estimate when trying to minimize response time of a distributed design problem. Basically, the traffic cost term captures the component of "response time" on a global scale that is expected in a distributed system. Part of the problem is to minimize the overall weighted number of hops in the system between the users and the servers.

In light of this, we see that the mirrored ex nihilo solver problem is NP-Hard. The problem of designing effective locations for ex nihilo server mirrors is intractable, and so will need to be treated by the same (or similar) techniques as those used in ex nihilo.

The modifications of ex nihilo needed to handle the server mirroring problems for parallel ex nihilo deployment will probably require a few months of code and algorithm development. The problem of using ex nihilo to find out how a mirrored set of ex nihilo solvers should be deployed is definitely an interesting problem.

Evaluating Recommendations from Multiple Optimizers: Time Stamps

There does need to be some coordination, and evaluation, of recommendations made by the optimizer, since mirroring data brings up issues of stale data, and data coherency. There will need to be some way to evaluate conflicting recommendations from parallel optimizers, based on some measures of trust.

Clearly, the oracle analyzing recommendations for state changes will need some kind of time stamps from the optimizers operating in a distributed environment. This needs to be more than just the time of recommendation by the optimizers involved in the conflict. It should also include time stamps, for each optimizer, of all data elements used by that optimizer in making its recommendations. A recent recommendation from an optimizer using stale data may be less reliable than an earlier recommendation from a competing optimizer using more recent data.

Evaluating Recommendations from Multiple Optimizers: Reliability and Connectivity

In addition to time stamps needed to evaluate measures of trust from conflicting optimizer recommendations, we should also try to evaluate how reliable a recommendation is, based on the failure probabilities of all system components of interest (hosts, routers, hubs, switches, links, etc.). Since these failure probabilities are key inputs to ex nihilo, and absolutely required to discuss survivability in any meaningful way, then this information will be readily available to all running optimizers (ignoring for the moment the stale data problem).

Consider a set of hosts, grouped together by locality, response time, failure rate, cost, function, or other measure of grouping. Since a key design feature of Ultralog is Survivability, we will need information on failure rates of the system components. We are going to relate the failure rates, and other statistical parameters, to some "Overall" probability of failure of a task we are trying to perform.

The optimizer requires probabilities of failure for all network components as part of its input set. We can set these probabilities of success to 100% for perfect system components, or 99.999% for most commercial equipment and communications. In war, this is more of a judgment call. A General may think there is a 25% chance that a position could be overrun by the enemy in the next few days (possibly in the supply chain). If this position contains key assets of the computing infrastructure, like Ultralog, then we would probably want to set the probabilities of failure of links and nodes in that area to 25% (or at least higher than those areas with low measures of expected failure). The optimizer would be expected to come up with radically different task deployments and routings in a situation such as this. In tests of ex nihilo, we have found just this expected behavior. The tool redirects parts of threads as needed to route around estimates of "future" damage to communications, and future damage to nodes (high risk areas).

We are going into details on probabilities of failure for two reasons. The first reason, is that it is an input to ex nihilo. It is easy in the program to set defaults, and change on the fly during a demo, so that part is not a problem. The second reason, is that we can use this information in deciding how one "group of nodes", can communicate with another group of nodes, and with what reliability. Remember, we are still trying to get some measure for our trust in the projections of conflicting ex nihilo solvers in a distributed environment.

Once we know how the tasks and threads are performed, in one particular operational mode, or state, we can get some "estimate/measure/guess" on the trust in communications/connectivity between any single host, or group of hosts (mirrored ex nihilo solver agents), and any other single (or group) of ex nihilo users. This estimate, of how well a particular host (mirrored ex nihilo server) is connected to other hosts, is another component of the trust measure used to resolve conflicts in recommendations from a set of mirrored ex nihilo servers.

Definition of State, and Reliability:

The knowledge of a system in one state needs to be defined a bit more precisely. It also needs to be defined to allow us to discuss the difference between reliability, and probability of success in one operating state of the system. We are minimizing the probability of failure (pfail) in the ex nihilo solver for one particular operational state, or mode of the system. From the glass is half full point of view, we are maximizing the probability of success. It is important to note that this is for "one particular operational state of the system". By state, I mean that all task and router assignments have been made. This is where reliability comes in... and backups.

There are a number of components of reliability, like node failover, alternate routing in the network, and alternate procedures in general. These types of problems of designing reliability into a faulty network fall into a special field of mathematics known as probabilistic graph theory. For now, let's just consider link failure, and assume the hosts/nodes are perfectly reliable. It can be shown that a graph with probabilistic failures of nodes and edges (links) in the graph can be converted to a probabilistic directed graph with perfect nodes, and failures concentrated in the edges of the graph. Therefore in theory, we can assume faulty links with perfect nodes. I point this out to emphasize that there is no loss in generality by assuming a network with faulty links and perfect nodes. The details on this transformation will be left for later discussion, and is the primary topic of our 2003-2004 statement of work.

The area of network reliability has been around for a long time, with many publications. The term "reliability", within the context of "network reliability" aficionados (some famous books with that title), usually refers to the overall probability of nodes in a network to communicate with each other with a certain probability of success. To calculate this "overall probability of success", we include all backup paths and failover modes in the network in the final calculation of probability of success. This is important. It is the difference between probability of success of communications in the physical topology (reliability), and the logical topology (routes defined by a particular set of messages, or packets). In large complex systems, with many alternate routes for messages, the difference in probability of success in communications for one particular message, in one state of the system, is much less than the overall probability of success (reliability) where we sum the probabilities of success of ALL of the different routes the message can take.

Network Reliability is #P-Complete:

The problem of network reliability is in another class of computational complexity, known as #P-Complete (Number P Complete). In the first section in this set of comments on ex nihilo, I discussed the NP-Complete class of problems, like the famous TSP problem, and some of the decision problems in ex nihilo. I then brought up the difference between decision problems (yes/no), and their related optimization problems. The "optimization versions" of the NP-Complete "decision problems" were said to be NP-Hard. We then pointed out that the difference in "hardness" between satisficing and optimizing is related to the difference in hardness between NP-Complete and NP-Hard. Both are classically intractable (appear to scale exponentially), although a good solution to the optimization problem automatically gives us a good solution to the satisficing or decision problem. In some sense, although both are believed to be intractable, the NP-Hard problem is "harder" than the NP-Complete problem.

There are a number of problems associated with knowing the size of the set (cardinality) of all possible solutions to a problem. To correctly calculate network communications reliability measures we need to find all backup paths for a given set of source and target nodes involved in messaging. As mentioned before, the reliability involves the sum of all possible network paths and failover modes. This is a famous, and difficult problem in network mathematics. We therefore see that a reliability calculation requires knowledge of all possible solutions to the problem, which in turn requires that we know the total number, or cardinality, of possible solutions (cardinality of the solution space). The enumeration of all backup modes needed to perform a reliability calculation is #P-Complete, which means that we have reached yet another level of intractability at the heart of Ultralog.

I don't want to get too far off topic, but remember the "NP" in "NP-Complete"? Although a rigorous statement requires that we define the problem in terms of something called a Non-Deterministic Turing Machine, a good "working definition", is that problems in "Class NP" can have a candidate solution verified in a polynomial number of steps on computer. That means that if somebody gives us a guess, or candidate solution to a problem (like a recommendation from ex nihilo), then we can verify that it is a solution in polynomial time. It says nothing about the difficulty in finding the guess, or candidate solution. For example, if I specify a particular solution to the TSP, by specifying a particular tour of the N cities, then I can verify if it is less than the maximum tour length TMAX in the problem by simply adding up the lengths of all of the path segments that make up the Traveling Salesperson tour. I can clearly perform this calculation, and verify the yes/no status of the solution, in a polynomial number of steps. Therefore, the TSP is in class "NP". The proof that it is NP-Complete is quite a bit more difficult.

We are discussing class NP, since it is "currently believed" that the #P-Complete problems are OUTSIDE of class NP (this is a famous unsolved problem in mathematics). This has important implications for both our reliability problems, and the use of network reliability to get some measure of trust for mirrored ex nihilo server agents in a distributed environment. The #P-Complete nature of the reliability problem means that even if we are given a good solution for a problem, that is supposed to meet the minimum specified reliability levels, we cannot verify that this is a good solution in a polynomial number of steps.

The topic of reliability, as compared to the probability of success of a specific operational state, is in our proposed statement of work for 2003-2004. I bring it up now, since it is a part of the answer to the question of how ex nihilo would operate in a distributed fashion, and how conflicts in recommendations should be handled from parallel sources of mirrored ex nihilo server agents.

Using Reliability to Generate Measures of Optimizer Trust:

Recall that a key problem in operating multiple ex nihilo solvers on the net, is trying to find a method of estimating ex nihilo reliability. We do not want the solver to be a single point source of failure, so we need multiple ex nihilo servers on the net. Since the net is faulty, and the multiple solvers may have varying degrees of old information, then we need to expect different solvers to make different recommendations for a distributed design.

To get a probabilistic measure of trust for a mirrored ex nihilo server, we would like to use the reliability of communications between the server in question, and the set of hosts on the network from which it requires input/output. If the node does not have reliable communications with other nodes, and has stale information about the nodes and links it is trying to optimize, it cannot reliably be counted on to perform calculations in performance and reliability.

If a systems optimizer is deployed in parallel, in a distributed network, with multiple copies of the optimizer running on a selected set of hosts on the network, then we gain reliability of the overall system, but at the price of additional complexity. We now have to make decisions on which recommendation to choose from the set of mirrored optimizers. The optimizers will have different connectivities, frequencies and priorities of use, time stamps, reliabilities, and other metrics, that will be used by the oracle in making a state transition. If the most recent recommendation also comes from a node with better overall reliability, the decision is easy. In conflicting cases, like a recent recommendation from an ex nihilo solver that is poorly connected (unreliable), then perhaps a compromise is called for. A compromise might involve choosing the solution state that is topologically (and managerially) closest to the current state. This falls under that famous rule of medicine "first do no harm". Asking a large number of sysops and network engineers to change their systems can be an effective tie breaker in major system design decisions involving conflicts.

The reliability is just one example of trust based on connectivity. There are a number of metrics of connectivity that could be implemented to measure how well an ex nihilo server is connected to the other nodes that use or contain ex nihilo inputs and outputs. A combination of time stamps, and optimizer connectivity/reliability, and other measures of trust would be used by the oracle to make decisions in cases of conflicting recommendations from the set of mirrored ex nihilo servers.

When a conflict in state change recommendations is found within a set of ex nihilo servers, it would be a reasonable operational policy to have the state decision oracle notify the conflicting set of optimizer agents to refresh their data inputs, and update the calculation.

Conclusions:

The use of an optimizer like ex nihilo is possible in a mirrored environment, although there will need to be carefully developed procedures for resolving conflicts in state change recommendations made by multiple optimizers. Some measures of trust for the parallel optimizers include time stamps of recommendation, time stamps of data used as inputs to the various optimizers, and how well the optimizers are connected to their sources and users of data, in terms of reliability.

The issue of how well a node, or group of nodes is connected to other nodes or groups, forms a natural transition to the topic. In the next section, we will discuss connectivity and reliability within the framework of a hierarchical or clustered approach. We will also discuss the "No Free Lunch Theorem" (for fun). There is now a brilliant mathematical proof for what most of us thought was "common sense" (for algorithm fanatics).

4.0 Hierarchical Modeling, No Free Lunch Theorem

In this section, I wanted to discuss some of the suggestions made during our workshop for solving problems in a hierarchical, or clustered fashion. Although discussions in our workshop focused on hierarchical modeling, the actual discussion centered around techniques that might better be described as clustering techniques. We will draw a distinction between clustered and hierarchical solvers, and discuss the conditions under which either or both of these techniques might be applicable. We will also discuss the "No Free Lunch Theorems", and explain how it relates to selection of algorithms (e.g., Simulated Annealing versus Genetic Algorithms).

No Free Lunch Theorems:

During our workshop, there was a question on whether we should use other techniques to solve the core "math" part of the problem. For example, techniques like genetic algorithms, tabu search, and even mixed integer linear programming, are occasionally used for some types of intractable problems. This is an important question, so I wanted to expand on my comments made during our workshop.

Some problems, like the Traveling Salesperson Problem, have excellent solutions for some subcases of the overall problem. For example, there are many good solutions for TSP problems, where the cities are in the plane, using Euclidean metrics, in special situations. There are other good solutions to the more general TSP that appear to be effective in many situations. It is therefore quite reasonable to ask whether there is some "best solution technique or algorithm" to be used in some of these optimization problems.

We frequently hear from various practitioners that "simulated annealing is better than genetic algorithms" (or vice versa), and that Tabu Search or some other technique is better than another technique. For the most part, most people that have worked with algorithms know that some techniques work well on one type of problem, while other techniques work better in a different problem type. Some algorithms do not scale well. They work well for small problems, but are useless for larger problems.

Some techniques only work well when applied to large problems. For example, you probably wouldn't use simulated annealing for simple routing decisions that have excellent polynomial time solutions by other techniques.

There has been some research in mathematics on how general algorithms perform in combinatorial optimization, when averaged over the space of all possible problems. Now this is not a light read, but I will give a reference to a set of theorems known as the "No Free Lunch Theorems", which address just this problem.

The basic results of the theorem are that when averaged over all objective functions, the choice of algorithms is basically a wash. So we might be able to find a faster (or slower) solution by switching to a pure genetic algorithm, or other approach for intractable problems, but there is no validity to the statement that "simulated annealing" or "genetic algorithms" are better in the general case.

Ex nihilo is an annealing solver, that uses a number of classic techniques in network mathematics and optimization for some of the subproblems, and has a "gene pool", that is probabilistically sampled when making decisions about the transitions between candidate states in the solver. It also has a greedy search solver that can be used to make fast starting guesses to the overall problem. So for now, it is a mix of a number of techniques that seems to work. Now if you have some good code for part of the problem, let's add it to the mix of techniques in the current solver.

If you would like to read more about the "No Free Lunch Theorems", then check out the technical report below from the Santa Fe Institute. The use of the word "theorem" rather than "conjecture" in the title should tip us off to the fact that there exists a mathematical proof of the statement. This is the theorem I mentioned during our load balancing breakout session in Feb. 2001. Have fun reading.

Reference to "No Free Lunch Theorem"

"No Free Lunch Theorems for Search", David H. Wolpert, William G. Macready, Feb. 23, 1996. SFI-TR-95-02-010

It is certainly possible to evaluate alternative approaches to solving some of the mathematical problems at the core of Ultralog. We already have tasks in the Statement of Work that do just that, namely, to investigate other techniques and approximations for some of the subproblems in the distributed agent problem. The goal is to find some fast techniques for getting a starting (or restarting) solution to the known intractable subproblems at the core of Ultralog,

Clustered Modeling:

During our Load Balancing breakout session (and in the main session), there was some discussion of modeling the entire Ultralog system in a hierarchical fashion. This was suggested as a way to attack some of the intractable problems within Ultralog, and get solutions in a timely manner.

Most of the discussion centered around the possibility of using a load balancer to adjust loads on a local basis, so I will start with discussing the use of ex nihilo to solve "local problems", involving groups of nodes and links.

In the previous section, I brought up the "No Free Lunch Theorem" for optimization. Depending on the structure of the overall Ultralog system, the use of clustering techniques may or may not be a useful approximation. In a very real sense, this is another case of this theorem. Some data sets, topologies, applications, etc., might have excellent fast solutions by judicious use of clustering and hierarchical techniques, while the application of these techniques to other problems may be the fastest way to find a truly horrible solution. So let's start with a discussion about clustering, and where and how it might be useful.

Consider an example situation where we have Ultralog, or any distributed system, mounted on four large data centers, located in Wash DC, Rome, Seattle, and London. Each data center may have multiple hosts/nodes. We might expect that we could break this problem down into four separate sub problems.

If a server in the Rome cluster needs information from any of the other three clusters, then it is not possible to truly separate the Rome cluster and solve it independently of the other clusters. A query from a Rome server to a server in London will place both traffic loads and CPU loads on the nodes and links in the London cluster. Now if the queries from Rome to the other clusters are infrequent, then we would expect that we could approximate the Rome problem as being independent of the rest of the system. When we have multiple frequent exchanges of information between the Rome cluster and the other clusters, then we should expect that any attempt to "optimize the Rome cluster" to possibly degrade performance in the other clusters, and perhaps degrade the overall global system.

A good "experimental proof" of this is the Internet. A common problem would be the case where a "backhoe" in Chicago digging a ditch, takes out part of the Internet backbone. When an outage happens, our routers adjust to the outage, and traffic is adjusted to other carriers and sources to account for the data link outage. The shifting of traffic places a burden on other parts of the net, and so on.. In many cases we find problems in communications, in other countries that are not even close to our "Chicago backhoe event". So to some degree on the net, everything impacts everything else on the net. Just like street traffic in Seattle, or Chicago, in traffic jams.

Matrix Representation of "Connectivity/Use"

We would like to represent the node system as a matrix. For each row and column we will attach a node name. If a node requires communications and processing by another node, we will put a non-zero entry representing the "strength of connection" between the two nodes. For a matrix "M", and two nodes labeled "i" and "j", we will set M_{ij} = some positive number when the two nodes require interaction, and Mij = 0 otherwise. Note that the actual numbers plugged into the matrix element Mij might depend on CPU requests from node i to node j, traffic (bytes/sec) from i to j, reliability of the i-j link, and other metrics.

Returning to our example, we will define four sub-matrices W, R, S, and L, for our computing centers in Wash DC, Rome, Seattle, and London. If Rome is truly isolated, then there will be nonzero entries among the Rome nodes, and the matrix R is a good representation of the local Rome computing center. If there are connections between Rome and London, then the interactions between these two centers can only be described by the full matrix M, since the Rome matrix R only has dimensions (rows and columns) for the servers in Rome.

We can define the full communications matrix M in terms of these sub-matrices as follows:

Communications Matrix for Wash DC, Rome, Seattle, and London.

$$\mathbf{M} = \begin{matrix} W & x & x & x \\ x & R & x & x \\ x & x & S & x \\ x & x & x & L \end{matrix}$$

The off diagonal elements "x" measure the strength of interaction between data centers. In the same way the sub-matrices W, R, S, and L represent multiple nodes, the values "x" also represent multiple nodes, and are themselves sub-matrices. The degree to which these "x" numbers are small is the degree to which the problems of four data centers are "separable". In the ideal case of separability, all of the x values are zero, and the matrix takes on a form known as "block diagonal":

Full Block Diagonal Matrix M:

$$\mathbf{M} = \begin{matrix} W & 0 & 0 & 0 \\ 0 & R & 0 & 0 \\ 0 & 0 & S & 0 \\ 0 & 0 & 0 & L \end{matrix}$$

Of course real systems are seldom block diagonal. For systems with small off-diagonal elements, then we can analyze the system as though the separate clusters were independent. This would result in considerable speedup of the solver.

Designing to Enhance "Block Diagonal Systems":

In the workshop I gave a demonstration of a system design problem with several "computing centers" that had strong off-diagonal matrix representation. This is the hardest case for the solver, since the problem is non-separable, and requires full global optimization of a larger system. One possible use of ex nihilo might involve designing a system which increases the block diagonal representation of the system. Since the ex nihilo solver moves tasks around, it has the freedom to arrange task/host assignments to maximize this block diagonal form. The problem is basically related to the problem of database partitioning, which is frequently modeled as a problem in "graph partitioning" in a graph theory representation.

To use ex nihilo effectively in this case, it would be necessary to first use the tool in a "design mode", where a system would be built to enhance locality, or "block-diagonalness" of the overall system. Once the system is built, it would then be possibly to run ex nihilo in real time, applied independently to the clusters discovered by ex nihilo during the design phase.

The use of ex nihilo to self discover block diagonal modes of operation would require some updates to the solver. To the objective function described earlier, we would need to add another term that measures block-diagonalness. A fast and easy way to do this is to add a term that would add an extra measure of "goodness" for architectures that perform multiple tasks on the same LAN or subnet. A more general approach would involve having the solver define "computing centers" as collections of nodes, and minimize the remote access among data centers. This is certainly possible, although it requires more discussion with other Ultralog developers before providing an estimate of development costs for ex nihilo.

Hierarchical Modeling:

Once we have some idea of how well a global system can be modeled/optimized as a collection of independent clusters, we are ready to discuss how such models might be used in a truly hierarchical sense.

Suppose we have a system in approximate block diagonal form. In this case we can model (approximately) an entire cluster of nodes as a single node, and study/optimize its interaction with the rest of the environment (other clusters). In this case, we would be looking for large scale statistical measures of traffic in/out of the cluster, and find some measure of optimality for the global system.

Even a non-block-diagonal system could be studied this way, although the global optimization problem (assignments, routing inside a complex cluster, etc.) would be turned off. In ex nihilo, this would be an easy task, since we could define the entire computing center as one "super-host", instead of multiple LAN's and hosts exchanging information. All one need do is move all defined task assignments within the cluster to the super-host, and then see how this super-host interacts with the rest of the world. All tasks within the cluster could be treated as communicating on the same host, with modified latencies and CPU burn times to account for the fact that we have moved everything onto one node. We could abstract this even further, and come up with approximate response times needed for the cluster to respond to the rest of the world, and get considerable speedup. To the extent that we are not interested in the details of how the cluster (super-host) acts internally, this would be an effective way to embed clusters into the overall Ultralog model, and get answers about the global system. This could be done within the current version of ex nihilo, since it would be a data problem, and not a solver problem.

During our workshop, I gave a demonstration of an advanced feature of ex nihilo called "Topological Reduction". When you view the network representation of ex nihilo, the full network view gives a fairly involved picture. There are Server Nodes, LAN nodes, and Routers, in addition to the links that connect things together. The topological reduction removes any node that is not associated with message forwarding, so basically we just see the overall topology of the network. This is actually a great tool for analyzing the global system without getting buried in server details. It gives a network engineer a snapshot of the things network engineers are interested in (links, routers, LANs). This would also be a candidate for using ex nihilo for hierarchical modeling. Starting with the full physical topology, one could do a topological reduction, and set single "super-hosts" at key points of the topology to approximate an underlying LAN communicating with its environment. The reduced topology could also be used to choose larger topological sub-chunks for approximation.

Conclusion:

We started the discussion of the "No Free Lunch Theorem", which roughly stated means "some algorithms work well here, some there, so there is no such thing as a best algorithm in general". This formed a natural transition into the discussion of hierarchical and clustered modeling. The ability to treat some parts of the system as independent clusters may result in considerable speedup, although it may come at the expense of degraded global performance. This degradation may be extreme, depending on how tightly or loosely coupled the clusters are. It may be possible to use models of individual clusters as single "super-hosts", to gain some speedup in the solver by treating the system in a hierarchical fashion. We also discussed how ex nihilo could be modified to design systems that operate in a predominantly block diagonal mode, which would improve the quality and reliability of modeling specific clusters as independent objects.

In the next section, I want to discuss "Failure, Reliability, and Survivability" in a bit more detail, and focus on Survivability. We have already discussed some background issues in Failure and Reliability, so we will build off of this work.

5. Failure, Reliability, and Survivability

In the third section in this series, I gave a description of reliability, and made a distinction between reliability of communications between a pair of nodes in the physical topology, and the probability of success of communications between two nodes in one particular state of the system. We also defined a state as being the operational mode of a system after all task assignments and routes have been completely defined. For communications, we compute the reliability by multiplying the probabilities of success for all possible routes of the message. We have also pointed out previously that a system with node and link failures can be modeled as a system with only link failures, without loss of generality.

The reliability problem, when rigorously defined, is known to be #P-Complete. The implications of this #P-Completeness are that if we are given a recommended solution, by some oracle that recommends solutions, then it is not possible to even VERIFY that the proposed solution is valid in a "polynomial number of steps" (although this is a famous unsolved problem in mathematics). This is in marked distinction to some of the NP-Complete problems at the heart of Ultralog, which have solutions that can be verified in a polynomial number of steps (although finding the solution may be exponentially difficult). Although many of the Ultralog problems are NP-Hard or NP-Complete, they are not *necessarily* #P-Complete. Many problems in reliability and survivability of distributed systems, when formally stated, are #P-Complete.

In this section, we are going to expand on the previous discussion of reliability, and bring up a new problem, Survivability, which is a primary focus of Ultralog (similar definitions after transformation). The survivability problem is also #P-Complete, which like reliability, takes us to another level of computational complexity, "believed" to be beyond the complexity of NP-Complete... at least we can check our guesses (from an oracle) in polynomial time for NP-Complete problems. We can't find the solution (so far) to NP-Complete solutions in polynomial time, but if given a solution guess to a problem, we can at least *Check* the solution in polynomial time, and see if it is a valid solution. In contrast, the #P-Complete problems, so far, can not even be checked in polynomial time, and are currently believed to be outside the "NP" class of problems. For Ultralog, this means that in very complex systems, with complex routing, that even checking the reliability estimates of the system is expected to grow exponentially difficult in the size of the system.

Reliability Revisited:

The reliability problem in network communications is the problem of calculating the probability of successful communications between a source set of nodes and a target set of nodes, in a faulty, physical network, or topology. This means we look at all possible routings for the source and target node set in question.

Since each set of specified routings defines a logical topology (as discussed in the first section and in our load balancing breakout session in Feb. 2001), then the reliability problem requires us to sum over all possible logical topologies definable in the physical topology. This hard problem is scheduled for 2003-2004, and is known to be #P-Complete. Recall that the #P-Complete problems are related to counting the number of all possible solutions to a problem.

The #P-Complete problems frequently surface in probabilistic calculations. In some of the NP-Complete "satisficing" problems (constraint satisfaction), we only need to find one solution that satisfies the constraint set. It may be difficult to find that solution, but we only need one. To perform reliability calculations, we need to sum over all possible solutions and connectivities in the solution space, which is a much "bigger" problem. Remember that probability is all about counting, so the #P-Completeness of some problems in probability should come as no surprise.

In the Ultralog problem, we are going to have failures at nodes and links, so our reliability calculations must include all possible options of node failover and alternate routing. To design a robust, reliable, survivable system, we are going to have to design both primary modes of operation with low failure probabilities, and backup/failover modes with low failure rates. So the design of a reliable system requires that we design a robust physical topology, as opposed to a single operational mode (logical topology). We need to design our physical system with a rich set of backup and failover modes of operation and routing to meet the overall "system reliability" measures of interest.

Survivability:

In this section I will discuss the topic of "Survivability", as compared to reliability and failure in a single mode or state of operation. This term is perhaps a bit more loosely defined in the literature compared to reliability, so I will discuss the problem from the "math-geek" perspective. To be specific, I will discuss survivability as defined in the "bible of computational complexity", namely, the book "Computers and Intractability, a Guide to the Theory of NP-Completeness", by Michael R. Garey and David S. Johnson. If you would like to read more, the chapter and verse for the problem is [ND21], entitled "Network Survivability", which I have rephrased in English.

In real world problems, we frequently encounter situations with "extra stuff" thrown into the problem definition. A key task of those working on algorithms used to solve real world problems is to separate the "extra stuff" from those parts of the problem that truly require special attention, or special techniques. For example, in the definition below of survivability, we are going to study a situation of "total collapse" of the system as a distributed system. It should be clear that in real world problems, we may be interested in "varying degrees of collapse". The core "hard part" of the problem remains the same. Although the problem is based in probabilistic graph theory, I will state the problem in English to reach a wider audience.

Consider a distributed system of hosts and data links connecting the hosts. The system may be connected in any arbitrary topological manner, and tasks (agents, etc.) may be arbitrarily distributed among the hosts. To operate as a distributed system, we need the hosts to communicate among themselves, so of course the hosts need to be operational. Furthermore, we need the data links to be operational in order to function as a distributed system.

For this example, we are going to investigate the situation in which the system has failed to survive as a distributed system. Perhaps some of the nodes are still operational, but the nodes cannot pass information, or interact, with other parts of the system, since the links connecting working nodes have failed. In every sense of the word, the system has failed to operate, to any degree, as a distributed system. The system has failed to "survive" as a distributed system. It has undergone complete failure in the distributed sense, because no working host can communicate with any other working host.

We are interested in finding the overall probability "q" that the system has failed to survive as a distributed system. To solve this problem, we will assign a probability p(x) for each object "x" in the distributed system, where x can represent any of the hosts or data links. Furthermore, let us assume that the failures of the objects are statistically independent, so the failure of a particular node or link has no impact on probabilities of failure of other nodes or links.

To begin analyzing the complexity of problems like this, they are stated as "yes/no" questions, called decision problems. An English statement of survivability is as follows:

> **Survivability Question (no communications):**
>
> "Is the probability q or greater, that for ALL data links in the system, the system will have failures of the data links, or failures at one of the end points of the data links?"

In the problem above, we are asking about the probability that the system has failed to survive as a *distributed* system. In this situation, any data link in the system has either failed, or has nodes at its endpoints that have failed. No part of the system can communicate with any other part of the system. The survivability problem as described above, is in the class #P-Complete, the same class that contains the reliability problem.

In the Ultralog system, we may have shades of gray, like situations with "60% node and link failure, and with 30% of all functioning threads remaining alive". As long as these shades of gray are well documented and precisely defined, then we can try to answer questions on "Survivability".

Ex Nihilo Solver:

The current optimizer is minimizing the probability of failure of a single operational mode of the system. That is an important point. The current solver searches throughout the set of task assignments, and physical topology (all logical topologies), for a mode to operate that "minimizes" the probability of failure of a task, subject to the min/max constraints on probability. The output is a logical topology (choice of routings) embedded in the physical topology (complete faulty network), and assignments for the tasks to hosts. This output (routings and task assignments) is the solver's best estimate of a solution at that point in time (that annealing step).

Our Statement of Work for next year, is to modify the tool to handle problems in reliability and survivability. The reliability problem has been around the phone companies for many years, so there are a number of classic techniques for good approximate solutions to reliability modeling. We have also planned some modifications to the ex nihilo solver that should allow the program to be used in a "constructive mode", to build a reliable system.

Ex nihilo is great for finding optimal (or good) solutions in a physical topology, and will attempt to find states least likely to fail. If the state fails, then the model could potentially be used to find the next state that is least likely to fail, and so on. The tool is currently designed to handle the single job of finding the optimal state of the system. We would like to enhance the solver to do a "greedy" search for backup modes of operation.

Designing Reliable Systems:

The next addition to the tool will enable it to find a reliable system configuration, as opposed to optimizing the current operational state. To build a reliable (or survivable) system, the designer is faced with a more difficult problem than deciding the current operational state of the system. The designer (or real time designer/agent), is faced with decision like where (and how) to place backups and failover plans into the system. In a sense, this is the difference between the NP-Complete and NP-Hard problems addressed this year, and the #P-Complete problems scheduled for 2003-2004. Basically, we are going to combine some classic reliability techniques, with some modified search techniques in ex nihilo, to create a tool that can tell us both the best current operational state, and tell us where to place our failover nodes and failover routes. A system *planned* for reliability and survivability will have a much better chance of finding reasonable operational states, when change of state decisions are forced on the real time optimizer (Chaotic environment). So next year, the program should tell us both how to operate the current state, and how to "minimize" future failure by generating carefully planned backup hosts (failover hosts), and carefully planned failover routes (which puts constraints on our physical topology).

Ex Nihilo "Gene Pool":

During the workshop in Feb. 2001, I gave a demonstration of ex nihilo, and gave a very brief introduction to the ex nihilo "gene pool", which was a small window that opened up next to the main solver window, and was labeled "Genetic Aggregate". We have now discussed enough in the last few sections that we are ready to talk about just what this genetic view is doing in the model, and how it can be used in the design of a reliable, survivable, Ultralog system.

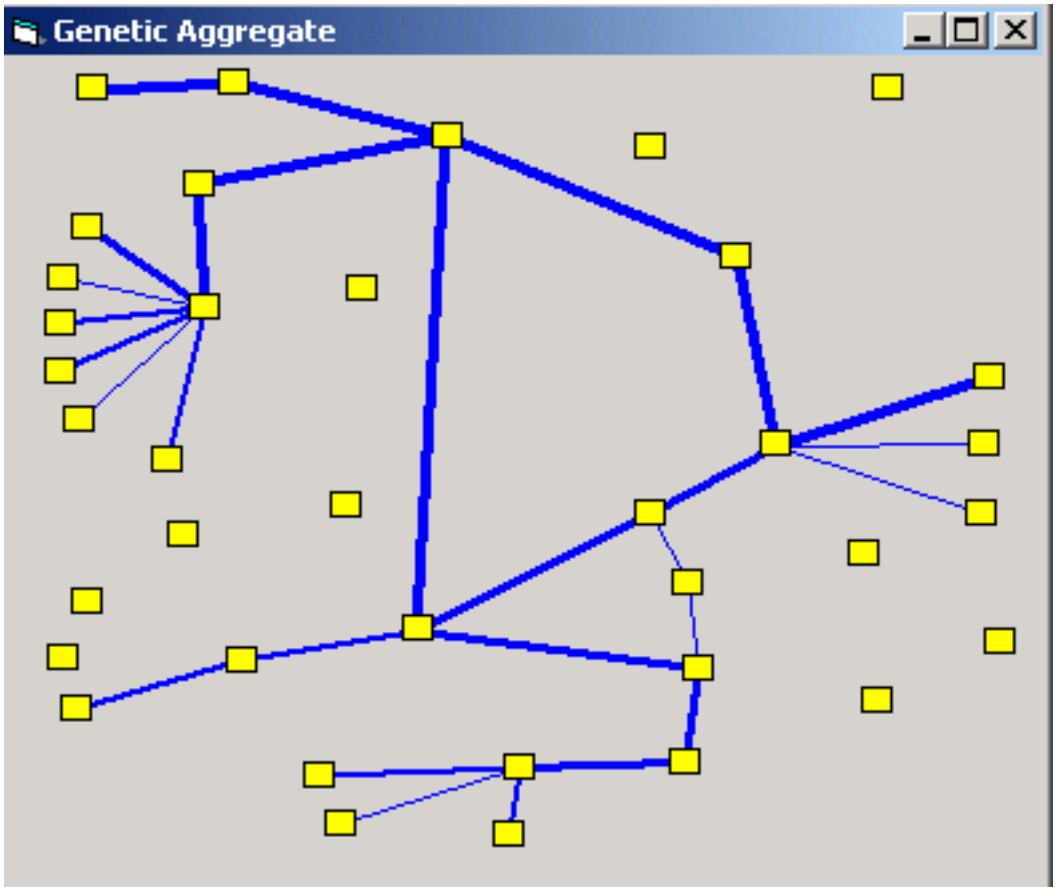


Figure 2. In the genetic window, we see a probabilistic view of all failover plans in the network. Lines that are thick are used by many more solution types, which become backup and failover plans in the system.

During the solution process, ex nihilo tracks all used paths throughout the physical topology, and makes a statistical estimate of the frequency of occurrence of each path segment in the set of all possible solutions (whether optimal or not). The "genetic aggregate" is a probabilistic view of all path segments, spliced together and plotted in a single window. In the GUI, the links connecting the hosts in this genetic view are weighted by thickness of the line segment for each link. The path segments that appear more frequently in the solver (higher probability), are drawn with thicker lines. This makes it easy to visually pick out links that are common to the greatest number of solution states.

I mentioned during the talk in Feb. 2001 that these thicker lines, in some sense, represent physical architectural backbones to the overall design. Clearly a failure in one of these backbone components will impact the greatest number of possible solution states, and so relates directly to our notion of reliability and survivability. A network designer would be very interested in the "thicker lines" in the gene pool, since as any network engineers know, a failure of the backbone will absolutely wreak havoc to the overall system.

There is another important point to be made about the ex nihilo gene pool. Besides using it as a valuable source of options for state transition, it also serves as our first view of what the required physical topology should be. Basically, the gene pool is a probabilistic projection of what ex nihilo thinks is the recommended physical topology (as compared to the logical topology, which is ex nihilo's projection of the current best operational state).

As a last point, I want to bring up the issue of reliability and survivability, and operating ex nihilo in a truly distributed mode, with mirrored ex nihilo server agents throughout the net (discussed in the third section in this series). In the third section, I discussed how the mirrored ex nihilo servers have different views of the world (due to communications failure), and how this could result in conflicting recommendations from the set of distributed solvers. Since each solver has its own "gene pool" view of the world, then it is reasonable to consider fusing the data from the set of parallel gene pools, and use this fused genetic view of the world to come up with a more realistic view for the genetic aggregate, which we now see is a probabilistic view of what ex nihilo thinks is a good, reliable physical topology for the distributed system.

Conclusion:

The issues of reliability and survivability are scheduled for 2003-2004. It is important to start thinking about these things now, since we need to understand as early as possible how all of this can be built into an integrated Ultralog.

In the next section, I wanted to cover the topics of memory and disk modeling, and the network issue of multi-homing, which relates to hosts having more than one connection to the network (e.g., one box with two network cards).

6. CPU, Memory, Disk, and Multi-Homing

This section is going to be a refreshing break, and an easier read, compared to the last few sections. We are going to cover some hardware topics, like CPU and Memory (modeled in ex nihilo), and Disk (not modeled). I wanted to point out that disk IO is not modeled in the current tool. It could be added, (few months??), but our efforts in the past have focused on CPU, since it was probably the most difficult and popular problem, along with cost and failure. The vast majority of our classic simulation models, even in distributed database modeling, focus on CPU, bandwidth, and latency as the primary bottlenecks in user response time (real stuff the users can see).

Our systems design tool, ex nihilo, was built as part of our research collaboration with Hitachi Corporation., where we focused on tools and design techniques for large distributed systems. Our team used classic simulation techniques for small systems analysis, but found a need to iterate over some difficult combinatorial problems to find good designs. The Hitachi team built a loop around the classic discrete event simulation tool used to model the distributed system. For very small systems, it is a reasonable way to go, but of course, even a single discrete event simulation of a distributed system can take hours...

Now some of the intractable problems discussed in the last few posts are going to need to be sampled thousands, or millions of times to get a reasonable solution, so an iterated simulation could take, at the minimum, tens of thousands of hours on fast workstations. Not a promising outlook for some of the intractable problems in database design, let alone solving general problems of task assignment and optimal routing, and reliability ...

From the Boeing side of the collaboration, we had the "ex nihilo" project, which had focused on intractability in distributed systems design. The systems designers were faced with difficult design problems in general, and many of these problems were intractable. The ex nihilo tool was built to solve design problems that were clearly beyond the "simulation in a loop" approach, and find good global solutions, and major architectural features, that could be later refined by more detailed discrete event simulation of special situations.

Queuing Network:

Since the use of classic discrete event simulation (hours) at each point of a combinatorial optimization problem (10,000+ iterations) is not feasible, we used some analytic approximations from queuing theory, and modeled the system as an object known as a queuing network. The queuing network label is a good description for what it does. It is basically a network of "servers", where a task passes through the network from one server to the next, and waits in line (queues) at each step of the overall sequence of subtasks required by the distributed task or job being studied.

As an example of a queuing network, a simple request for a web page might consist of the user submitting a web page url (with appropriate protocols) to a "data link server" (like a WAN). Here the WAN is modeled as a "server", with time spent in waiting, and time spent in transmission. Once the request has left the WAN server, it moves on to a web server (real host), where it spends more time waiting for service by the CPU, plus the time spent being serviced by the CPU. It could then be modeled (not currently modeled in ex nihilo) by entering a "disk" server, with the disk queuing time plus disk service time adding to the overall length of time spent in the task (thread). After leaving the "disk" server, the task in progress would reenter the "WAN" server, spend a bit more time, and return to the user with the web page of interest.

There are commercial products (like SPE*ED) that model systems via queuing networks in just this way, without resorting to lengthy simulations. The problem, is that the queuing network products did not handle combinatorial optimization (other than hand iterating in a blind manual search). So we developed ex nihilo as an "open queuing network", with a simulated annealing optimization engine, and using fast analytic approximations for the queues used in the individual "servers" used in the queuing network.

For most distributed systems design problems, the analytic queuing network is a good approximation for estimating response times. We use this fast queuing network, and other fast approximations, in building objective function during each step of simulated annealing. The robust simulated annealing solver gives us the assurance that the optimization will converge, and gives us the ability to easily adjust our objective function based on response time, probability of failure, and cost. I will discuss the details of the queuing model when I cover the topic labeled "Non-Markov Processes". This is a very quick overview of what things look like under the hood of the ex nihilo solver.

Closed Queuing Network:

Since the predominant use of ex nihilo has been in public or corporate systems, we were usually concerned with random arrivals of short jobs, in a system with many users. In a batch job situation, similar to the current Ultralog "Send OPlan" jobs, we are going to have make some modifications to the queuing modeled used for response time. To model batch jobs, the standard approach is to use a queuing model where the job submission rate from the user is equal to the job response time from the system. This is called a "Closed Queuing Network", and requires iteration of the basic open queuing model. The iteration begins with a best guess (or long time, 24 hours) at the system response time, and use that to get a starting job submission rate (1 job per 24 hours). At the first iteration of the solver in the closed queuing network, the system is totally unloaded, so the response time for "Send Oplan" job will be fast (relatively, like 12 minutes). Use this new response time (12 min) from this iteration, as the input to the next iteration, so the job submission rate will be a little faster (1 job per 12 minutes). Repeat until self consistent, which defines the batch time response in a closed queuing network, with no wait between jobs.

Since the Ultralog application is at present a batch problem, then program modifications will be required when producing estimates of Ultralog run time or response time. The specific modifications involve converting the ex nihilo solver from an open queuing network to a closed queuing network. Initial experimentation showed high rates of convergence in the closed queuing algorithm. For example, convergence in job submission and job response times in the model converged to within three decimal places after 2-3 iterations of the closed queuing loop.

Disk Modeling:

Now that we see that ex nihilo is modeling system response time via a queuing network, it should be apparent what needs to be done to add disk modeling to ex nihilo. We would need to build a model of a disk as a "disk" server in the queuing network, and add it right before or after the "CPU" server. We would want to consider including things like "seek time", "rotational latency", and transfer rate. We would also need the amount of information read or written to disk during a particular subtask. In the same way that we built queuing models for our CPU model in ex nihilo, we could use queuing models for disks to add disk IO to the model if that was needed.

The implications on disk modeling is clear. For systems with heavy disk IO, where the disk read/write time exceeds the CPU time, LAN/WAN response times, and other times in the modeled task, then ignoring disk IO is a poor approximation.

The careful reader might have noticed that we stated earlier that the "disk server" could be added before or after the "CPU server" in the queuing network. Of course, real computers multitask, with disk IO sometimes taking place in parallel with CPU processing. And some tasks burn some CPU, read/write to disk, burn more CPU, read more, and so on. A rigorous treatment of disk IO would require a detailed knowledge of the operating system, and how the individual task is structured, queued, cached... so in general a very tough problem. To really get the best possible model of disk modeling, we need to return to simulation modeling, and hour++ simulations, and no ability to treat intractable problems like database partitioning, and task assignments to minimize response times and failure.

Memory:

Accurate memory modeling is extremely messy. As a general rule, it requires intimate knowledge of both the application and the operating system, so is hard to build into a generic modeling tool. One way to get a rough handle on it is to specify a fixed amount of memory per process, and choose that memory so that the process does not have to undergo significant paging. We can usually get an estimate on required memory for a resident process, so a basic memory model might consist of adding the memory requirements of all resident processes and reasonable allowances for data. If possible, we want *everything* to be in memory. We would probably also want to account for shared memory for some class of tasks that use common memory (to avoid double counting). If the total memory requirement is greater than the amount of memory in the host, it might be wise to reassign a task to another server, since the current server would be expected to be paging.

A memory model like this would not be that difficult to implement in ex nihilo, and would be quite a bit easier than adding a "disk server" to the queuing network as described above. The Aug. 15 delivery of ex nihilo had a memory model that had fixed constraints on each server, with no virtual memory or soft failure. Each agent in the Aug. 15 2002 delivery of ex nihilo is required to have a specified memory consumption. It was agreed during our Conference in Albuquerque that the memory supplied would be the high water mark of expected memory usage. If an overall solution cannot be found that falls within memory and CPU utilization limits, then ex nihilo produces a "solution not found error".

An upgraded version of the model has been proposed in which soft memory failure would be allowed. The solver would look for solutions with the smallest violations of expected memory limits of the system.

CPU:

The CPU model in ex nihilo is built to model multiprocessor systems, so it shows up as a "multi-server queue" in the queuing network part of the solver. I will talk about the queuing at the CPU in detail during the discussion of Non-Markov processes, a couple of posts from now.

Since ex nihilo was built as a design tool, it will actually search the space of routings, task assignments, and server configurations to minimize response time, failure, and cost. It has been tested, and could potentially will make recommendations for the number of CPU chips in each host in the final solution, subject to user specified limits. This is a useful feature for designing distributed systems that meet response time constraints at minimal cost. It was also added as a speedup measure to the solver to improve convergence for searches over multiprocessor systems.

Multihoming:

The last topic I wanted to cover in this section is the topic of "Multi-homing", which in distributed systems applies to the concept of hosts belonging to more than one network. Most hosts on the internet are singly homed, meaning they have only one network as their home network. For example, if you take a look at the back of your PC or workstation, you will probably find one and only one network connection or network card. In most cases this is fine, since the PC is not a mission critical component, so your PC is probably a "singly homed" host on the internet. There is no provision for communications failure, in the sense of having a backup network connection.

One occasionally sees multihoming on DNS servers (domain name system). A single physical host may be connected to two or more physical networks to improve reliability, or even improve network load balancing. In cases such as these, using the UNIX command "nslookup" (on some pc's too), will result in multiple IP addresses for the same domain name.

Consider the case of Ultralog.net. If we type "nslookup ultralog.net" at the UNIX prompt, we find that it maps to a single IP address, 4.22.165.13 (at time of writing), so a user typing "ultralog.net" is going to always go first to the site "http://4.22.165.13". There may be a load balancer at 4.22.165.13 that sends us somewhere else, but basically the domain name points to one and only one box initially. Now type "nslookup amazon.com", and we see "207.171.183.16, 207.171.181.16". This could mean that there are either two network cards on a single box for amazon.com, or possibly, that amazon.com is on two physical boxes for the initial entry point (with an armada of boxes behind the entry point to load balance traffic and server loads).

A single server homed on two or more networks is usually described as "well connected", since there are two or more distinct connections to the net. A key design point, is that important hosts in the system should be multihomed, if possible.

As an aside, a cheap form of load balancing is to have the same domain name point to two or more unique IP addresses, on two or more unique hosts or network cards. The "Bind" software (Rev. 8 or later), used by most of the net for DNS service, does a round robin approach to domain name resolution. Each time Bind receives a query for the IP address of a domain name (like amazon.com) it randomly picks one of the IP addresses assigned to the domain name. If the IP addresses are for different physical boxes, we have a cheap automatic load balancer of hosts. If the IP addresses are for different network cards on the same box, then we have the user request being routed through different networks (assuming the network cards are "homed" on different networks). For big companies like amazon.com, we probably have a mix of all of the above, mixed with some expensive load balancing hardware that goes beyond the free load balancing intrinsic to Bind.

I bring this up, since there was a question on whether ex nihilo had the ability to move agents to hosts that have multiple network connections. This was an excellent question. The short answer, at present, is YES/NO. The tool will search out the net for the task assignments that minimize response time (and failure and cost), so if that response time is minimum on a multihomed host, then the answer is "yes". The answer is "no" in the sense that ex nihilo does not show preference towards multihomed hosts. In some sense, multi-homing is a reliability and survivability question to be addressed next year. To build in the basic ability to give preference to multihomed hosts would actually be a fairly easy task, but is not currently in the schedule. The tool is certainly capable of modeling multihoming, which was demonstrated during the workshop. It does not, however, give preference to multihomed hosts. So multihomed hosts as a "preference" is easy. If we are interested in multihoming from a reliability perspective, then that will have to wait until next year.

I would like to make one final comment on multihoming with respect to network load balancing (in contrast to host based load balancing). If we want to split a specified fraction of traffic along two or more routes, then that can be handled within the current model by dividing the same physical load into two or more virtual loads, and defining the function calling architecture so that the loads take different routes. For example, we can model situations where 60% of traffic goes down one path, and 40% traverses an alternate path. A more difficult problem is deciding whether the split should be 60/40, or perhaps 55/45 in the optimal state. This is related to the mirroring problem discussed earlier. There is some partially stubbed out code in the tool for the mirroring problem, but it would take more time to include searches for optimal host mirroring and route splitting to multihomed hosts.

Conclusion:

It is important to recognize that the current version of ex nihilo addresses networking issues for the system, and CPU and memory issues at the host level. It does not currently address disk (other than at the cost level). A basic memory model to avoid paging has been added for the Aug. 15 2002 deliverable of ex nihilo. Adding a disk model would be more involved, but certainly no harder than problems already addressed in ex nihilo, and could use classic queuing models from the literature for disk systems. To "model everything" in minute detail is really a simulation task (to be discussed later), and frequently does not bring enough "additional accuracy" to warrant the effort. Even basic simulation models run for an hour or more, so are not realistic for real time use. The current approach of using an open queuing network for host modeling, and an annealing algorithm for the intractable parts of the search, are reasonable approximations for solving hard problems in distributed systems design in a reasonable amount of time.

A first cut at including multihoming preferences into the model would be fairly easy (days/weeks ???). A more involved approach to the use of multihoming considerations in task assignment and route planning really gets to the heart of the reliability problems scheduled for 2003-2004. The problem of finding "optimal" traffic splits for communications with multihomed hosts is related to the mirroring problem.

The next section will address issues related to "Recursion, Performance Modeling, and the Turing Halting Problem". This gets to the heart of the load modeling needed to create an accurate performance model. I will discuss in detail some of the data requirements for ex nihilo, along with some of the pitfalls. I am sure it will stimulate some discussion.

7. Function Definition, Performance, and Turing Halting Problem

In the last several sections, we have covered a number of design questions, and covered some theoretical aspects of the ex nihilo solver. In the next several sections, we are going to cover the inputs and outputs of ex nihilo.

Since ex nihilo has to model the way programs burn up resources on a network, its basic user input is a collection of functions that call each other, in a statistical sense, and burn up resources on the network links and servers.

In the workshop demo, I demonstrated a set of user initiated calls to a set of functions which were running on the user's workstation. These functions on the user's workstation will correspond to high level starting tasks performed by agents, users, or other process initiators and tracked in terms of response time, failure, fixed cost, and monthly cost. The functions in the initial set of calls are members of the set of all user-defined functions in ex nihilo. When any function is called, it is allowed to perform some very primitive tasks, like burn CPU on the box it is on, and possibly make calls to another set of functions (which adds to network messaging).

The function calls in the model are usually based on estimates of CPU burn for major user transaction types. A performance model usually *does not* model down to the micro level of an individual SQL statement in a user-database interaction, or line of code in a C program. A performance model *does* focus on jobs, tasks, and the moving of data. The most popular topics of interest in performance modeling are response times for jobs, and job failure rate.

Response Time:

The response time for a particular task in ex nihilo is based on the length of the longest running thread spawned by the task. So we start with a set of functions, that can call other functions, and track down all of the threads, in parallel, and find the length of each thread. Let's look at one particular thread, and see how it performs. In these threads, each call is generated at the end of the CPU service time, plus latencies and transmission times of the task. This model captures most simple processes, like a 3 tier calling architecture, although it would not capture complex threads, based on some final availability of selected other threads, which when all fully ready, open a latch to allow a new thread to be spawned.

Consider one thread, where we follow one particular line of function calls, until we reach a stopping point, or halting state. At the last function in the sequence of function calls, there are no further calls to other functions. halt. We get a time stamp for that thread, and log it as the longest thread. Do that in a loop over the calls spawned from the "initial set of user calls" or start button, until all calls have been traced, at which point you are done. Since the function tracing is explosively complex, be sure to keep the function calling depths and lengths small.

It is important to emphasize that we are building a performance model of a system, so a function call is defined as a job on one server that burns some CPU, uses some memory, and possibly sends calls to multiple other agents on other servers. at the end of the function call. Currently, ALL subroutine calls are made at the end of the thread, and there is currently no provision for delays between multiple function calls throughout a single thread. That looks possible, but is not in the plan at the current time. Other complex thread tracking strategies have been proposed.

It is important to emphasize that we are building a performance model. So in a complex problem like Ultralog, we might want to begin with some work on specific threads of interest. We would look for some obvious large users of resources, and try to understand their resource needs in light of current systems availabilities. It may happen that a large CPU user in a set of users or agents may be using a large amount of CPU in the beginning of the thread, and deliver information long before it is actually needed (brings back pert charts doesn't it?). If a specific agent, task, or subthread, is burning excessive resource on a task, without a pressing needed (like longest thread response time bottleneck), then it may be locking out or delaying other threads that truly are on the longest thread, or "critical path". This implies we could develop a solver that would, if it was supplied with some measures of expected task durations and rates, be able to delay or pause tasks on the critical path in real time during thread and system execution. This is an exciting prospect.

To estimate the length of this particular thread of calls, we need to first calculate the time waiting in queue before our first function can be executed, and then add the CPU burn time for the time it takes to actually service the function call (current version does not model disk response times). Once this is done, we pick one of the "subfunctions" called by the first function, find where the subfunction is located (which host), and send a message to the subfunction. Remember, we are doing this algorithmically, inside the solver. The message is a user specified message size, measured in bytes or other appropriate unit.

To model the network transmission time for the message, we use bandwidth and latencies, and a selection of standard or custom protocols for modeling packetization effects, like MTU, and protocol overhead. The time of arrival of the second function call in the thread is just the time spent on the first server, plus the messaging time between function calls. At that point, we follow this process recursively through the thread until we reach a halting state. The time to halting state for this thread is the run time for this particular thread. To find the overall response time from the first user initiated task until completion, we find the maximum thread time of all threads generated by the first user-initiated call.

Since the time spent on a server, for a given function call, is the sum of CPU burn time plus the time waiting in queue, then we need to estimate the time waiting in queue (we can measure CPU burn). The time waiting in queue is based on other jobs on the system, and their arrival rates. To estimate the overall job arrival on each server, we track all of the threads, and compute the loading contribution from that thread (call rate) to each server in the thread of calls.

Once we have an estimate of the loading on all servers, from all threads, we can use some fast (?) techniques from queuing theory to estimate the time waiting in queue. I will cover this in more detail in the section on non-Markov processes. The time in queue plus the CPU burn time, plus messaging time, gives us a good estimate on our primary user metric of interest, response time. Any other modifications, like adding disk effects, are easily added to the program architecture, since the main program logic in response time is tracking the threads.... The probability modeling is quite a bit more complex, so I will cover it later (2003-2004 topic).

Touring Halting Problem:

In light of the above, we see that part of the input to ex nihilo is a "user-defined program", comprised of a set of functions that call other functions. The ex nihilo solver follows all threads from the starting task, until an overall halting state is reached, where all threads have halted. We track the time length of the longest thread, so we know the overall response time for the main task at the top of the thread. As long as we don't have infinite threads, we are OK...

The set of input functions to ex nihilo, and the associated calling architecture, represents a program that is fundamental INPUT to ex nihilo. We need to know whether this set of functions and function calls, along with other data representing input to ex nihilo, will reach a halting state, or will it churn away forever, looking for an end to some infinite thread buried in the set of function calls.

During the design of ex nihilo, it was obvious that there was a famous difficult subproblem buried in the heart of the overall problem of tracking function calls to determine response time (and load the servers). Since ex nihilo uses user-defined functions in its performance model, as part of its input set to find a solution, the program needs to be able to answer the following question: "Is it possible for a programmer to write a program that will detect if another program reaches a halting state?" This is an extremely famous problem in mathematics and computability, known as the Turing Halting Problem. In the general case, the answer is NO. It can be shown that there is no way to develop an algorithm to determine whether another general program reaches a halting state.

In ex nihilo, the programs are not "general programs", but a very specifically defined set of function calls, so we can at least try to check some of the input data ;) The user defined functions are very primitive, and are only allowed to call other functions, or use resources like CPU on their current node. The call from one function to another is where the messaging model comes in.

When functions call other functions in the system, they send messages for traffic modeling, and burn CPU, and possibly track memory and disk IO in the future. In this primitive environment, all we need to do is track all threads, and look for recursion, to determine if the system will reach a halting state. If in a sequence of function calls (in one particular thread), we discover a function call in that thread that had been previously called in that same thread, then we have found an infinite loop in the input to ex nihilo. The end of the thread is reached when the function being called makes no further calls. At that point in the thread tracker, its thread-time is compared to the registered longest thread found so far, with an update to the longest thread tracker if required. Although we have no hope of solving the Touring Halting Problem in general, at least we can try in ex nihilo to check for infinite loops in the checking routines before the solver stage. For most multi-tiered systems, this allows an effective model of performance to be built.

Ex nihilo has a checker for the set of function calls used to define the user input to the tool, with a well deserved ;o). When you run the function call check utility, it tracks all function calls, throughout all threads spawned from the initial set of user tasks, and looks for infinite loops based on simple recursion, until it finds a problem, executes normally, or crashes due to memory problems in the calling structure in the algorithm. We need to work on that, but then again, it is the Turing halting problem, so we get some slack on input checking.

The function check utility will stop, and inform the user if it finds a recursive call in a particular thread (calls to the same function in parallel thread calls are not OK in current checker, but this is changeable). If the checker finds no problems with the function input, and all threads reach a halting state, then it lets the user know that the input is OK, and can be used by the search and trace parts of the solver. The only other option for the check utility, is if it runs forever (hard to measure). That can only happen if the set of function calls to ex nihilo is infinite. I don't see that happening soon on this project, ;)

Fortunately (?), the function calls forming the input to ex nihilo are primitive, with no internal logic. These are not the "general program input" problems of Turing Halting fame, so thread tracking is fairly easy (??? says the programmer). A given function has a finite defined set of other functions as part of its input, so all we have to do is write a program that follows all threads to termination. Since there are a finite set of these simple function calls (although combinatorially large), then the function checker will eventually stop. If the length of the calling stack is beyond virtual memory (including other large data structures), ex nihilo goes "poof", back into the vacuum, and crashes. It is ex nihilo's personal way of saying that this is a big problem, so try to simplify the performance model. Note that this overflow occurs during the checking stage, and not during run time.

Modeling Recursion:

How do we build performance models of recursive functions? To get an estimate of recursion, we can start with estimating the depth of the recursive calling series. Remember, each "call" is really "number of calls per second" in performance modeling.

Consider a very simple set of recursive calls, where function Fa on server Sa, is calling function Fb on server Sb. The function on server Sb will use some CPU and other resources, and call Fa by sending a message (Kbytes) to function Fa on server Sa. At this point, we have a recursive call. To build a performance model, we need to get some statistical estimates on the depth of the recursion. Even a simple 2 function, 2 node example, at the recursive level, can be a major task if the recursion is deep:

In general, we could start looking at subcases, and breaking them down statistically in the expected operating environment of Ultralog. The table below breaks down a simple function recursion by percentage of system case load:

20% Fa -> Fb

10% Fa -> Fb -> Fa

20% Fa -> Fb -> Fa -> Fb

25% Fa -> Fb -> Fa -> Fb -> Fa

10% Fa -> Fb -> Fa -> Fb -> Fa -> Fb

10% Fa -> Fb -> Fa -> Fb -> Fa -> Fb -> Fa

5% Fa -> Fb -> Fa -> Fb -> Fa -> Fb -> Fa -> Fb

100% Projected Load use (ignores server assignments)

Since the current tool does not allow any user defined recursion, these cases need to be modeled as a series of independent function calls. We need to rename (or clone) some of the functions in the ex nihilo input, to capture the recursion for a performance model.

In the statistical breakdown of load depth, listed above, our current performance model would need to model all of the listed groups (each line) of cases.

20% Fa1 -> Fb2

10% Fa3 -> Fb4 -> Fa5

20% Fa6 -> Fb7 -> Fa8 -> Fb9

25% Fa10 -> Fb11 -> Fa12 -> Fb13 -> Fa14

10% Fa15 -> Fb16 -> Fa17 -> Fb18 -> Fa19 -> Fb20

10% Fa21 -> Fb22 -> Fa23 -> Fb24 -> Fa25 -> Fb26 -> Fa27

5% Fa28 -> Fb29 -> Fa30 -> Fb31 -> Fa32 -> Fb33 -> Fa34 -> Fb35

In the above performance estimate of the recursive call, each of the functions can have unique inputs based on CPU and other performance parameters. In general, all functions are different, so the simple two function two host recursion expands into a large collection of unique functions. In many cases the functions will be simple clones of each other, with the same inputs and different names. This naming convention keeps the ex nihilo solver from getting trapped in infinite loops generated by recursive calls. In light of this, a primary requirement for the model is that recursion is translated into statistical cases based on recursion depth.

In general, modeling a recursive situation is extremely difficult when the recursive call is buried deep in the function calling tree (not a direct call, like above, but with intermediary calls to other functions before the recursion begins). In situations like this, it frequently means you are modeling things in too much detail. In some cases, this difficult recursion depth analysis may be the only way to find the depth of the longest thread, needed to find the overall response time of the main task spawning this recursion.

Modeling Function Calls:

To find a function/host assignment and set of routings that produces an optimal solution state, based on response time, failure, and cost, then we are going to need some statistical measure of resource used per task/server assignment. We will also need a statistical measure of function calls by a given function. For example, in a four function system (ignore servers for now), with functions f1, f2, f3, f4, we might have the following breakdown of how function f1 calls the other functions:

30% F1 => F2

250% F1 => F3

100% F1 => F4

380% Total function call rate!

In this example, function F1 calls F2 at 30% of the rate that F1 is called. In the second line, function F1 calls F3 at a rate of 250% that of F1 (F3 call rate is 2.5 times the F1 call rate). In the last case, we have function F1 calling F4 100% of the time.

This percentage breakdown of how functions call each other is a fundamental input to ex nihilo. Any reasonable performance model of a system is going to need estimates of expected tasks over the period it is intended to model performance. This functional breakdown, along with the tasks of finding the performance parameters of the functions (like CPU burn), is the reason it usually takes a *dedicated performance data team* to find inputs to performance models. <mode=whining>I have said since day 1, that a dedicated performance data team is needed to make major inroads to UltraLog. If I have do this data mining, nobody will build the solver </whining> Register as possible problem, and flag for action.

Function Definition in Ex Nihilo:

We are now prepared to list the data requirements for each function in the set of user-defined functions in the model input. For each function F in the defined set of functions, we need the following function properties as part of the basic ex nihilo input set:

1.) For each subfunction Fx called by F, we need the percentage of subfunction calls to Fx for each call to F. This is simply the ratio of call rates for the called function and calling function. For example, an agent or plugin may make 100 SQL queries to the same function, doing similar things on average from a CPU and memory perspective.

2.) For a selected class of functions that define the user-initiated functions (main agent tasks), we need to specify an initial calling rate (calls/sec). These user-initiated function calls are the major tasks that are traced for response time. The subfunction calling rates, that are based on spawned calls from the first level user-initiated tasks, are completely determined by the initial calling rate of the user-defined functions, and the ratios of spawned function calling rate to the rate of the caller functions, as defined in list item 1 above. For batch jobs, these calling rates are determined from a self consistent search in a closed queuing network.

3.) For each function call from F to a subfunction Fx, we need the size of the message sent from F to Fx, in units of Kilobytes.

4.) For each function F in the function set, we need the set of hosts (servers) on which the function F is eligible for execution. Note that eligibility does not imply that the solver will actually choose to execute the function on a particular eligible server. A function Fb called FROM a server Sa will be assigned to a server Sb based on the current state matrix. Although the server assignment for calls FROM a particular host are defined to be on one and only one server, this does not preclude the called function from running on other servers when called FROM other hosts.

5.) For each function F, and for each server S for which the function is eligible for execution, we need the amount of CPU burned during the execution of the function F on the specified server S. We also need to know the amount of memory that should be reserved for the agent.

6.) For each function F, and for each eligible server S, we need the fixed cost of installing the function on the server.

7.) For each function F, and for each eligible server S, we need the monthly cost of running the function on the server.

Possible Modifications to Function Definition:

Although the above situation covers a large number of usage cases, there are some modifications that might be of interest that are *not* in current Statement of Work:

1.) The function CPU burn rate is based solely on the server on which it is executed. A generalization would be to modify the model to allow for burn rates based on the calling node for a function, or on the function calling the function in question. For example, a function Fa in the model may have different CPU burn rates when called from Fb as opposed to calls from Fc, on average. In most cases, this could be modeled in the current tool as a set of different functions on a case by case basis, although it would lead to major growth in the number of functions used in the tool.

2.) There is no messaging latency for functions called from the same host. For example, if two functions Fa and Fb are running on server Sa, then a call from Fa to Fb will incur no message delays. This is usually a reasonable approximation, since calls to the "localhost" occur on the same server, making localhost latency small compared to other delays in the model (LAN/WAN latencies, etc). The current solver could be modified to include localhost latency.

Conclusion:

The load definition in ex nihilo is based on a set of primitive function calls, where each function is allowed to burn resources like CPU on the host computer, and make calls to other functions, which generates system messages. Defining the function properties and calling architecture is an art in performance modeling, and typically requires the resources of a dedicated data analysis team.

By carefully restricting the way functions are defined in ex nihilo, we were able to avoid serious problems in determining whether a given set of functions would reach a halting state. Since recursion is disallowed in the programs generated by the user-defined function set, then it is necessary to build a statistical view of recursion, and model the recursive series of calls as a statistical collection of non-recursive calls. While it is certainly possible to generalize the set of functions definable by the user, and use this set of generalized functions to simplify problem definition for the ex nihilo user, at some point we should expect additional problems related to the Turing Halting problem. It is the responsibility of the user to ensure that inputs to ex nihilo are well defined, and that these inputs, when used by the ex nihilo solver, create a problem that reaches a halting state.

We have also given a complete listing of all function properties needed to define a function. Some of the resource properties like CPU burn per transaction, and message sizes, will be discussed in the section on load testing. Other properties, such as the rates of user-initiated function calls (main tasks), and frequencies of calls to subfunctions, can only be estimated by careful data analysis of the problem.

I moved the "fault detection in networks" to the optional section of topics (after the dashed line). If there is interest, I will cover fault detection, which is another NP-Complete problem at the heart of Ultralog. For small systems it is easy, while for larger systems (and finding the location of the fault), it can be difficult. The agent/host optimizer could potentially be designed to choose a set of target and source nodes for periodically polling to determine system fault status.

8.0 Host and Data Link Definition

To define a problem in ex nihilo, a user needs to define the properties of the three primary objects in the problem: Hosts, Data Links, and Functions. There are a few other requirements for the objective function which we have discussed previously, and will review further in the next section (response time, failure, cost), along with a few run parameters.

This section will cover link and host properties, separated into the major categories of "MAIN", "CONFIGURATION", "COST", and "MISCELLANEOUS". We will begin by defining the Host properties, followed by the definitions for Link Properties.

8.1 HOST DEFINITION:

8.1.1 Host MAIN Properties:

Host Type: (Server, Shared LAN Hub, Router, Virtual Link Node)

A host in an ex nihilo design is actually just a connection point in a network, and may be one of three broad categories of hardware devices.

If the object has a "Server" Type, then it is allowed to send and receive messages, and compute functions (use CPU and memory), but is not allowed to forward message packets to other computers.

If the object is a "Shared LAN Hub" type, then the object is a shared LAN hub, and has the ability to broadcast messages to all other hosts connected to the hub, in standard Hub fashion.

If the object is a "Router" type, then it is a router, and has the ability to forward messages. A Switched Ethernet can be modeled with a router object and directed Link objects. A possible enhancement to simplify the modeling of Switched Ethernet, would be to create a "Switched Ethernet" type of object.

Host Name: (40 Character Text String)

The Host Name property is just the text name of the host object. For example, a node associated with processing paychecks might be labeled "Node-paycheck". Spaces are allowed in the Name property.

Host Number: (Integer, 1 - numhost)

Each host in the network has a unique positive integer, referred to as the Host number. If there are "numhost" hosts in the current design, then the host number must be between 1 and numhost. All IO, including flat file input and output is listed in order of Host Number.

8.1.2 Host CONFIGURATION Properties:

Host Background CPU Utilization: (%, Real*4)

This is an estimate of the background load on the server. For the Ultralog "Send OPlan" example, it should be a measure of percent CPU utilization after the app-servers and nodes have been launched, but before the "Send OPlan button" is pressed. In this case, we have an estimate of the background load of the system, before processing starts.

Host Max CPU Utilization: (%, Real*4)

In the current model of response time, the "Max CPU utilization" is based on queuing theory. A high level of CPU utilization will manifest itself in lengthy queuing at the CPU. When ex nihilo is used for optimizing response times, then it reject states where processors have abnormally high queuing (which results in long response times). The Max CPU Utilization constraint is frequently used by systems managers to place a safe limit for their system, like 80% Max CPU Utilization. Many single processor systems begin to show significant queuing at 80% level, with steep increases in queuing beyond 80%. Multiprocessor systems (like 16 CPU servers) can usually be pushed harder, approaching 90% utilization. The "flatness" of the queuing curve for multiprocessor systems can be easily shown using queuing theory. Once a multiprocessor begins significant queuing, the degradation in performance is severe, compared to single processor systems. Another way of stating this is that single processor systems give systems operators (and users) "fair warning" that the system is approaching saturation, while multiprocessor systems frequently give little warning as they approach saturation (like hitting a brick wall without notice). A careful response time model should carefully model the effects of multi-server queuing.

Host Number of CPU's: (Integer)

Each Host of the Server Type must have a defined number of CPU processors greater than zero. This field is ignored for Routers and Shared LAN hubs. For Server types, the number of processors is needed for our CPU queuing model of response time.

Host Max Number of CPU's: (Integer)

Each Host of the Server Type must have a defined MAXIMUM number of CPU processors greater than zero, which is the Max Number of CPU's property. This field is ignored for Routers and Shared LAN hubs. It is used during the "subspace iteration" part of the search to configure servers to obtain good response time at minimum cost.

Host Memory: (MB, Integer)

Each host is allowed to have a specified amount of memory (RAM), and is measured in Megabytes. The Memory property can be used in cost calculation, although not in the current Ultralog implementation of Aug. 15 2002.. A memory model enhancement has been discussed elsewhere.

Host Number of Disks: (Integer)

Each host is allowed to have a specified number of hard disk drives. The Number of Disks property is only used in the cost calculation. A disk model enhancement for response time and throughput has been discussed elsewhere.

8.1.3 Host COST Properties ($$$):

Host Installation Cost: ($, Real*4)

The Installation Cost property is the fixed cost of the host, including installation, in dollars, but WITHOUT memory, CPU, or Hard Disk. It is the price of a raw host, with floppy drive.

Host Cost per Month: ($, Real*4)

The Cost per Month is the monthly cost of maintaining the host. It should include regular labor, building lease, server lease, insurance, 24 support, software lease, etc.

Host Cost per CPU Processor: ($, Real*4)

The Cost per CPU Processor is the cost of an individual CPU chip.

Host Cost per Disk: ($, Real*4)

The Cost per Disk is the cost of an individual hard disk drive.

Host Cost per MB Memory: ($, Real*4)

The Cost per MB Memory is the cost of one megabyte of memory.

8.1.4 Host MISCELLANEOUS Properties:

Host Message Forwarding: (Integer: 0,1)

The Message Forwarding property determines whether the host is allowed to forward IP packets in a network environment. If the value of message forwarding is set equal to "1", then IP forwarding is allowed. If the value is set equal to "0", then IP forwarding is not allowed. Always set packet forwarding to "1" for Routers and shared LAN hubs. For servers, set the value to "0", unless you actually want a server to perform message routing service. For servers that both server functions and perform IP message forwarding, then this mixed host mode should be possible, but would require time reworking the data structures in the program.

Host Failure Probability: (Real*4: 0.0 - 1.0)

The Failure Probability property is the probability of failure of the host, in some (unspecified) user measurable time T. For example, 0.001 is a common failure rate for commercially advertised hosting services (non critical service).

Host Max Packets/Sec: (Real*4, Proposed, Not modeled)

The Max Packets/Sec property is currently not used, but does show up in the GUI and underlying data structures.

The value of the Max Packets/Sec property for a given host is the maximum number of packets that can be processed by the host. This is mainly of interest to router modeling and NIC card processing capabilities. For routers, the maximum number of packets per second is a primary measure of "router performance" in the performance industry. The model does currently do the required packetization calculations to account for MTU constraints (discussed in Link Definition section) and to properly account for message protocol overheads when sending messages. It also tracks packet rates through all hosts, but does not constrain traffic based on packet rate. Although packet rates are tracked during calculation, there is currently NO output making this information available to the user.

8.1.5 Host GUI Properties:

Host Color:

Host Text Color:

Host Position on 640x480 screen:

These are fairly obvious, and only used in the GUI. For Ultralog this year, we automatically pick some nice colors and positions for the servers.

8.2 LINK DEFINITION:

8.2.1 Link MAIN Properties:

Type: (Shared Hub Link or Directed Link)

A link in an ex nihilo design is a data link connecting two defined Hosts in the network. The link type must be either a Shared Hub Link, or a Directed Link.

A Shared Hub Link type is only used to model data links connected to shared hub based LANs. This may also be used for shared data links in a "daisy chain" type of LAN topology. In modeling information and traffic in a shared hub environment, the traffic is shared with all nodes on the LAN. The messages are broadcast, in typical shared hub fashion, so a message from one node on the hub to another node on the hub is actually broadcast to ALL of the nodes on the hub. The hub based links are mathematically bi-directional, although only a single link in the model is used to model bi-directional traffic (both directions have same bandwidth and traffic). By convention, a Shared Hub Link in ex nihilo must originate at the server, and terminate at the hub. This connectivity order is used in the model to simplify shared hub definition, and minimize the number of links used to define a shared hub network. The ex nihilo data check utility will warn the user if a link is connected to hub in the wrong direction (physical direction from hub to server, as opposed to the required server to hub connectivity).

A Directed Link is the other basic Link Type. Links that have the Type property set to "Directed Link" are directional. A switched Ethernet hub can be modeled as a collection of Directed Links connected to a Router Type of Host object.

The current version of the tool allows one and only one link connection between two hosts. It is important to note that this does NOT mean that there is one and only one path between hosts, but rather that only one physical link is allowed between hosts. A possible enhancement would be to allow more than one physical data link to be modeled between two hosts.

Link Name: (30 Character Text String)

The Link Name property is just the text name of the data link object. For example, a link connecting server S1 to server S2 might be named "S1-S2". Spaces are allowed in the Name property.

Link Starting Node: (Integer, 1-numhost)

The Link Starting Node is the Host Number at the start of a Directed Link. For Shared Hub Links, this must always be the Host Number of the Host connected to the Hub.

Link Ending Node: (Integer, 1-numhost)

The Link Ending Node is the Host Number at the end of a Directed Link. For Shared Hub Links, this must always be the Host Number of the Hub.

8.2.2 Link CONFIGURATION Properties:

Link Latency: (Seconds, Real*4)

The Link Latency property is the standard one-way latency of the data link, measured in seconds.

Link Bandwidth: (Kbit/Sec, Real*4)

For directional links, the Bandwidth property is the one-way bandwidth of the data link. For Shared Hub links, this is the bandwidth rating of the Hub object (assuming the bandwidth ratings of the cabling and network cards supports the hub bandwidth rating).

Link Background Traffic: (% Bandwidth, Real*4)

The Link Background Traffic is the % of total physical bandwidth that is used by other background processes in the system.

Link Probability of Failure: (Real*4: 0.0 - 1.0)

The Probability of Failure property is the probability of failure of the Link, in some (unspecified) user measurable time T.

Protocol Overhead: (Bytes, Integer)

For a given TCP/IP protocol, this is the number of Bytes of Overhead information associated with a given packet. The Overhead section contains information like IP addresses, "Path MTU Discovery Flag", and other information not directly associated with the actual user data in the packet.

Maximum Transmission Unit (MTU): (Bytes, Integer (Real*4 internally in code))

The Maximum Transmission Unit (MTU) is the maximum size of the data area at the network layer of the TCP/IP protocol stack. For packets with data sizes exceeding MTU, then the model will fragment the packets into multiple smaller packets to meet the MTU constraint. The sum of MTU plus Protocol Overhead is equal to the physical size of the packet or datagram.

For real networks, there are basically two ways of packetizing a message. If the "Path MTU Discovery" flag is on in the protocol header, then the two servers involved in communications are tested along the path defined by the current router settings, and the smallest MTU in the path is determined (discovered). This value of MTU is then used by the sending server to appropriately size packets before transmission. If Path MTU Discovery is off, then the packet fragmentation is performed, successively, at each router along the path. The packets are reassembled at the receiving host.

In ex nihilo, we do not know the path, since part of the solver's job is to find appropriate router settings that minimize response time, failure, and cost of the overall task set being modeled. To estimate overall protocol overhead associated with a given message along each link, ex nihilo takes the original message size, and packetizes the message for that particular link. In most cases, this is a good approximation, and is exact for systems with homogeneous values of MTU for all network links, or in cases where the MTU values form a monotonic decreasing set of values along the path. It does introduce error when we have alternating values of MTU along the links in the path, like MTU(big)->MTU(small)-MTU(big).

The packetization effects due to MTU constraints on overall message overhead is usually small (a few percent). The model could be enhanced as follows to get a better approximation. If the router settings are fixed (and not found by ex nihilo), then the calculation could be updated to perform the exact calculation for any type of network by implementing the required packetization for the specified route. This could be done with or without Path MTU Discovery enabled, although it would require another user input to indicate the value of the Path MTU Discovery Flag. If ex nihilo was used to find optimal routes, then it could be possible to first find the optimal route using the current packetization procedure in the model to find an "optimal specified route". This route could then be corrected (as above) to account for MTU effects along the current estimate for the optimal specified route. Clearly, the MTU corrections could in theory force a different alternate route to be the estimate of the optimal specified route. This change in estimated optimal route is unlikely to occur, since MTU corrections are by their very nature small. Since ex nihilo already makes a first order estimate of MTU effects, then the differences in routing generated by second order effects are likely to be extremely unlikely. The other way to approach the problem, is try to include MTU effects directly in the routing solver. This could be messy, and would require additional development of the routing algorithms. In most cases, the current approach to handling MTU constraints is probably "good enough".

8.2.3 Link COST Properties:

Installation Cost: ($, Real*4)

The Installation Cost property is the fixed cost of the Link, including installation and hookup charges, and equipment costs, in dollars.

Cost Per Month: ($, Real*4)

The Cost per Month is the monthly cost of maintaining the data link. It should include regular labor for network maintenance, monthly lease, 24 support, etc.

8.2.4 Link GUI Properties:

Link Line Color:

Link Line Width:

Save Link Icon Position:

These are fairly obvious Link properties, and only used in the GUI. For our demo this year, I will just pick some nice colors and positions for the links.

Conclusion:

In this section, we have covered all of the key properties associated with Link and Host definition. We have also given some comments on how the ex nihilo tool could be modified to handle cases not currently addressed.

In the next section we will cover some of the key run time constraints associated with the objective function (like max response time), along with some of the other parameters used during a typical run.

9. Miscellaneous Run Parameters, Background Modeling

In the last three sections, we have covered all of the properties associated with the three major types of objects in ex nihilo: Hosts, Links, and Functions. In this section, we will cover the remaining parameters needed to define an ex nihilo run.

Some of these run parameters, like MIN and MAX constraints on the objective function, were covered in the second post of the series, "Thrashing, Satisficing", so this will be a short review of these parameters. Other run parameters will be discussed in a bit more detail.

Each of the miscellaneous run parameters will be described with a short paragraph where we will discuss the meaning of the parameter, and its acceptable range of values. Where appropriate, we will also describe suggestions for future modifications, and whether or not the suggestion is in the current Statement of Work.

Background Modeling:

Host Background Load:

Assume a function "F" requires a total amount of CPU ***T*** on a server "S". The servers in ex nihilo models have no intrinsic "speed" property. There is no clock speed in an ex nihilo host, so everything is defined in terms of the CPU time it takes to execute a function on a specific server. The time ***T*** is the TOTAL amount of CPU resource required to complete the execution of function F, regardless of background. For example, our function F in the function definition section of the model is defined to use a total of "***T*** CPU Seconds". In a time sliced or queued environment, the "clock time" will probably be quite a bit longer than T to complete the task.

Consider a case where we have a 30% background load (of unknown origin), with no other processes running on the server. In this case, we have a processor with approximately 70% as many "useful CPU cycles" as the rated performance of the CPU chip (ignoring context switches, etc.) To account for this in the function definition section of ex nihilo, we would need to increase the value of ***T*** to ***T***/0.7 to account for background load.

Since ex nihilo uses queuing theory for estimating times, we need to take a step back, and consider just what ***T*** is in the model. The value of ***T*** used in the function definition section is the time it takes to SERVICE the function F with a dedicated CPU processor on host S. In queuing theory, this time is known as the "Service Time". Suppose the rating of the CPU is M MIPS. In the time ***T*** (in seconds), we have the following expression for the number of instructions executed in time ***T***. The estimate on number of instructions ignores a lot, but it works for our discussion.

M = MIPS rating of one CPU chip

T = CPU time required to perform F (CPU Seconds, as measured in lab)

Instructions_Total = Instructions required for task

$$\Rightarrow$$

$$\mathit{Instructions\_Total} = \mathit{M} * \mathit{T}$$

Now, let's look at the effect of adding a background load, like 10% of CPU used for other services not specifically accounted for in the thread and agent tracking in ex nihilo (OS tasks, safety factors...). Suppose we have a background utilization of ***U*** (fractional, not percentage). In this case, we have effectively reduced the MIPS rating of the chip by a factor of **(1-*U*)**, so lets call the effective MIPS rating ***Mnew***.

$$\mathit{Mnew} = (1 - \mathit{U}) * \mathit{M} = \text{"effective MIPS rating"}$$

What we now have, is a function with the same number of required instructions, that must be run on a processor with effective MIPS rating of ***Mnew***, so we find the following expression for the new service time ***Tnew*** on the server with background utilization ***U***: To find our

Since we are required to perform all of the Instructions required to perform the job or task (***Instructions_Total***) then we find our new response time ***Tnew*** including background effects as follows:

$$\mathit{Instructions\_Total} = \mathit{M} * \mathit{T} = \mathit{Mnew} * \mathit{Tb} = (1 - \mathit{U}) * \mathit{M} * \mathit{Tb}$$

$$\Rightarrow$$

$$\mathit{Tnew} = \mathit{T} / (1 - \mathit{U})$$

This is how background load is treated in the CPU model in ex nihilo. It is this value ***Tnew*** that we use for "effective service time" in our queuing theory model of response time, based on the user specified background load for the node.

Link Background Load:

The explanation of Link Background Loads is similar to the above.

Here are the variables used in link background loads:

B = Physical Bandwidth of link

Bnew = Effective Link bandwidth

U = Background Link Utilization (fractional)

$$\Rightarrow$$

$$Bnew = B * (1 - U)$$

The difference between the CPU factor of **(1-*U*)** in the denominator for servers compared to the **(1-*U*)** multiplicative factor for links is due to the fact that for servers, ex nihilo has no concept of server speed, so we modify the loads on the server (***T*** => ***Tnew***). For data links, we do have a real speed rating (bandwidth), so we modified the actual link bandwidth. Of course, we could have played the same game, and modified all message sizes in the function definition section to account for link background loads, but that is a lot more work for a simple work around.

Miscellaneous Run Parameters:

Weighting of User-Initiated Functions: (Real*4)

The current tool tracks a single class of user-defined function on a single host for response time, failure, cost, and monthly cost, as described in the Function Definition section (Section # 7). This needs to be generalized to allow tracking/optimization of multiple user-initiated functions, originating on either single or multiple hosts. These user-initiated functions correspond to the set of all agent tasks that are tracked for response time, failure, and cost.

To track and optimize multiple agent tasks, there needs to be weighting factors to weight the relative importance of the tasks. For each task in the traced task set, a real number weight must be assigned to the task. The relative importance of each task's contribution to the overall objective function is given by the ratio of the task's weight to other task weights being tracked. For example, consider two tasks, labeled TASK1 and TASK2, that are being tracked with weights ***W1*** and ***W2***. Let the objective function contributions for each of these two tasks be labeled as ***OBJ1*** and ***OBJ2***, and let the overall objective function be labeled as ***OBJTOT***. Then the objective function is given by the following expression:

OBJTOT = W1 * OBJ1 + W2 * OBJ2

These task weights might actually be associated with subthreads of the overall Ultralog program, and perhaps weighted according to an estimate of their overall impact on response times. Regardless of the values of weights used in the demonstration, the ex nihilo tool will have the ability to arbitrarily set weights to any values chosen by the user community.

The problem of choosing relative weights is a difficult, and ill-defined problem, as it is with any objective function used to optimize a system with different goals and objectives. For most cases, the ex nihilo user is more interested in meeting a set of "satisfactory constraints", which are well defined. The solver will search for operational states that first satisfy the constraint set, and secondarily optimize performance based on the relative weights defined above.

A suggestion would be to define constraint categories on response time, etc., for each agent task of interest, and assign relative user-defined weights that are roughly equal, in the range of 1.0 to 10.0. The categories would then be used to assign tasks to major categories of importance or preference. A first estimate of category weightings would be to scale the weights by 10%, so a three category system might have weights like 1.0, 1.1, and 1.2. In this example, the solver would give a mild preference to the third category for optimization, with weighting of 1.2. Assigning large user-defined weight differences, like 1.0, 100.0, 10000.0, although well defined, might heavily skew the solver to favor the largest weighted category, and ignore the other categories.

Ideally, the weights should be chosen in consideration of the actual values of the objective function components. In our example of a two task problem, if a typical value of ***OBJ1*** is "7.0" and a typical value of ***OBJ2*** is "7000.0", then we might want to assign weights with values of ***W1*** = 1.0 to task one, and ***W2*** = 1.0E-3 to task two, to give an approximation for equally weighted tasks. A simple equal weighting (***W1***=***W2***) would result in the second task giving a much greater contribution to the overall objective function.

Since the user has no a priori knowledge of the objective function, then the generation of weights based on the objective function is really out of the user's control. This problem also comes up in giving weights for objective function components based on response time, failure, fixed cost, and monthly cost. To handle this problem, ex nihilo uses a dynamic objective function, which changes during the solution, and performs an internal scaling of response time, cost, etc.. When all user defined weights are equal, then each weighted component of the objective function is equal. Using this technique, the user of ex nihilo only needs to set the relative user weights (e.g., ***W1***, ***W2***) to obtain a solution which ex nihilo (internally) scales to yield the desired result. This is addressed in more detail in the next section dealing with response time, failure, and cost measures.

The use of a dynamic objective function based on the current solution state simplifies program input for the user. All a user needs to do to have equally weighted agent tasks is set all weights equal (e.g., ***W1***=***W2***=***W3***...=1.0). The solver will automatically create a dynamic objective function to produce a reasonable estimate of equal weighting.

This weighting problem also comes up in the setting of weights for objective components based on response time, failure, cost, and monthly cost (described below).

Weighting of Objective Function Components: (Real*4)

As discussed elsewhere, the primary objective function components are Response time, Probability of Failure, Fixed Cost, and Monthly Cost.

Since the tool tracks user-defined functions for response time, failure, fixed cost, and monthly cost for a given task, there needs to be some form of prioritizing or weighting the individual components that make up the objective function. The problem of comparing cost (in dollars) to response time (in seconds) is at best ill-defined. As with the problem above (Weighting of User-Initiated Functions), the constraints in the problem are probably the easiest way for a user to find good solutions to the general problem of optimizing with respect to conflicting goals, in different units.

Consider a single task, with an objective function contribution ***OBJ1*** (e.g., a single agent case, as described above). The optimizer calculates ***OBJ1*** based on the response time, failure, fixed cost, and monthly cost for this task as follows:

OBJ1 = Wt * Rt * T + Wp * Rp * P + Wfc * Rfc * Cfc + Wmc * Rmc * Cmc

where:

Wt = user defined weight for response time

Wp = user defined weight for failure probability

Wfc = user defined weight for fixed cost

Wmc = user defined weight for monthly cost

Rt = Automatically generated Response Time scaling factor (internal to ex nihilo)

Rp = Automatically generated Failure scaling factor (internal to ex nihilo)

Rfc = Automatically generated Fixed Cost scaling factor (internal to ex nihilo)

Rmc = Automatically generated Monthly Cost scaling factor (internal to ex nihilo)

T = Response Time of task being tracked

P = Probability of Failure of task being tracked

Cfc = Fixed Cost of task being tracked

Cmc = Monthly Cost of task being tracked

Consider a typical case, where fixed cost is on the order of $10,000, while the response time is on the order of 1 second. It should be clear that in this situation, that if ALL weighting factors above are equal (internal and user defined weights), then the objective function ***OBJ1*** above would have its major contribution from fixed cost, with response time contributions being negligible. In a typical scenario, fluctuations in cost would be on the order of thousands of dollars, compared to fractions of a second in response time, so the solver would spend most of its time trying to minimize cost, and ignore response time. Clearly not what the user wanted.

To simplify the problem of user input, ex nihilo uses a "dynamic objective function", that is scaled internally to produce reasonable results for reasonable values of user-defined weights. To generate the dynamic objective function, the solver takes a statistical sample of solutions during run-time, and calculates a moving average of the four basic components (response time, failure, fixed cost, monthly cost). It then takes these averages, and generates a set of INTERNAL weights, ***Rt***, ***Rp***, ***Rfc***, and ***Rmc***, so that the internal weights, multiplied by the averages, are equal. This internal scaling ensures that the four internally weighted components have the same approximate values, on average. For example, in our case of an average fixed cost of $10,000, and average response time of 1 second, the internal representation might set ***Rfc*** = 1.0e-4, and ***Rt*** = 1.0. These internal scale factors are adjusted (moving average) as the solution progresses.

Using this dynamic objective approach, the user now only needs to set ***Wt***=***Wp***=***Wfc***=***Wmc*** to define a problem where we have approximately equal weighting of Response Time, Probability of Failure, Fixed Cost, and Monthly Cost.

In light of the above, the main requirement from the user for weighting factors for objective function components is a set of user-defined weights for each task in the set of tasks being traced. The values of the weights are best kept in a range of values near 1.0 to 10.0 to produce reasonable results reflecting user preferences.

Initial Starting Solution:

At the start of the solution process, ex nihilo requires an initial starting solution, or seed solution, to begin the search. It helps if the starting solution is a "good solution", but it is not required. The starting solution is not even required to be a physically reasonable solution, and it may violate all of the user-defined constraints for response time, failure, and cost. As long as it represents a well defined state of the system, the solver will be able to search the solution space for near-optimal solutions. If the starting solution is a good solution, then it will take less time to converge to a near optimal solution, so it is to the user's advantage to choose a good starting solution if one is available.

There are two possible ways to use a starting solution. In the first method, the user selects a totally random state. This is the method I demonstrated at our Feb. 2002 workshop, and is the required method when there are no previously stored "good solutions" available to the user. The second method, is to use a solution that has been previously generated and stored by the solver. The program has a utility for saving and restoring valid states of the system. Any of the states saved by this utility are valid solutions to read into the solver as the Initial Starting Solution.

Min and Max constraints on Response time, Failure, Cost, and Monthly Cost: (Real*4)

The details and discussion of the constraints on objective function components have been presented in the second section of this series, entitled "Thrashing, Satisficing". The constraints have the same meaning for all four components of the objective function.

The Max constraint is the dividing line between acceptable and unacceptable solutions. For values less than the Max limit, the solution is acceptable. For values greater than the Max limit, the solution is unacceptable.

The Min constraint is the dividing line between the set of "equally good" solutions at or below the lower limit, and solutions which have variations in goodness. For components less than the Min constraint (e.g., response time < TMIN), then the solver does not attempt to improve the quality of the solution along that dimension of the solution space. For components greater than the Min constraint, the solver will continue to search for better solutions.

As an example, consider the response time limit of TMIN = 0.1 seconds. If the solver finds a state with response time equal to 0.09 seconds, then it will consider this solution to be "equally good" compared to a solution with a response time of 0.001 seconds. In this case, even if a solution with 0.001 response time exists, the solver will not attempt to find it when the current response time is 0.09 seconds. If the solver finds a solution with a response time greater than our minimum limit (TMIN = 0.1), for example a response time of 0.2 seconds, then it will continue to look for better solutions, with response time less than 0.2 seconds, until it finds a solution with a response time less than or equal to our minimum limit, TMIN = 0.1 seconds.

Subspace Iteration: (Check Box)

The Subspace Iteration part of the solver is used in "design mode", to find server configurations that satisfy response time constraints at minimum cost. During a subspace iteration, the solver will vary the number of CPU processors defined on each host, and choose an optimal server configuration.

For run-time use in optimizing an existing system with well defined hardware, the Subspace Iteration check box should be left unchecked.

Cost/Failure Routing Corrections: (check box)

The "Cost/Failure Routing Corrections" part of the solver is used during run-time to generate an optimal set of routing matrices that also optimize with respect to Cost and Failure. Recall that normal routing decisions are based on the number of router hops between communicating hosts, which is a much simpler problem than the problem of generating a set of router matrices that minimize Response Time, Failure, Fixed Cost, and Monthly Cost.

For the demonstrations this year, it is recommended that the ex nihilo user turn off "Cost/Failure Routing Corrections" (unchecked). Since we are starting with a fixed logical topology (this year), there is no need to add corrections to the search based on routes optimized for Cost or Failure. Turning off the Cost/Failure Routing Corrections will speed up the ex nihilo solver.

Screen Refresh Rate: (Seconds, Real*4)

The Screen Refresh Rate is a GUI feature. It is the time that elapses between refreshes of the of the main graphics screen. It has no impact on the solution.

State Viewing: (GUI, Radio Button)

The State Viewing option is a GUI feature. During the solution process, the solver hops around from one state to another state. The annealing state is the current estimate of the "best solution", as defined by the annealing search. Since the solver may hop out of the best state found in a search, then the annealing state may or may not be the actual best state found in the search. It can be shown that as the solution progresses, then the annealing solution will converge with 100% probability "in distribution" to the global optimum. Since real solutions are usually terminated before the global optimum is found, then the current annealing state may actually be worse than previously discovered states. To account for this, the solver stores both the annealing state, and the first three "best states" found during the solution process.

The GUI allows the user to choose between viewing the annealing state, or the current state of the system during the search. The current state of the system is mainly used when using the ex nihilo Trace Utility. The Trace Utility provides a complete view of any current state, and includes a function traceback utility, and volumes of output on events associated with that current state.

During solution iteration, leave the State Viewing option on the Annealing/Run page set to "Annealing".

Annealing Temperature: (Real*4, Scroll Bar Input)

The Annealing Temperature is used by the ex nihilo solver as part of the Simulated Annealing algorithm. A discussion of how simulated annealing is used within ex nihilo is left for future documentation. Although the topic is extremely theoretical, a quick overview follows:

The simulated annealing algorithm is a technique that has been around since approximately 1983 as a popular search technique in optimization, although early forms of the technique can be traced back to the work of Metropolis in the 1950's. The technique is based on the analogy with annealing of metals in physics/metallurgy. The process of annealing a piece of metal consists of heating the metal up to a high temperature, and slowly cooling the metal after reaching the high temperature. The high temperature puts the metal into a microscopic (quantum mechanical) state of random orientation of the metallic "crystals" (domains) within the metal. As the temperature is slowly lowered, the crystals are allowed to hop into and out of various states of alignment. At low temperature, the crystals tend to be aligned, which is a lower energy state for the system.

By analogy with the process of annealing in quantum mechanics and metallurgy, the collection of states in a combinatorial optimization problem form a thermodynamic system. In the theory of simulated annealing, we draw an analogy between the "energy" of a physical system, and the value of the objective function used in optimization. Frequently in the literature, the terms "energy" and "objective function value" are used synonymously.

At high temperature in an optimization problem, the system is allowed to hop around randomly, from one state to the next state, with no preference given to states of lower energy (objective function value). The algorithm that determines whether a proposed state transition will be accepted is known as the Metropolis algorithm. At high temperature, all state transitions are equally acceptable, so the algorithm behaves like a random search. As we lower the temperature, the transition to states with lower energy will be accepted with higher probability (as determined by the Metropolis algorithm). At very low temperature, the algorithm only accepts transitions to lower energy states, and only moves "downhill". This mix of "random" search at high temperature, and "pure downhill moves" at lower temperature, allows the search technique to move across the hills and valleys of the objective function, and avoid getting trapped in local minima (or maxima). This is what makes simulated annealing a good choice for global optimization.

I am well aware of the fact that the above description is a bit sketchy (to say the least). The underlying theory of simulated annealing draws on a number of fields like Thermodynamics, Information Theory (entropy), Inhomogeneous Markov Chains, and others. In addition to drawing on these fields, the selection of an appropriate annealing schedule also relies on pattern recognition techniques to spot "phase transitions" in the solution space. As the temperature is lowered, it is important to know when to slow down the cooling rate to avoid getting trapped in a local minimum.

For ex nihilo to operate properly, it requires a procedure for lowering the annealing temperature from high temperature at the start of the search, to low temperature at the end of the search. The current tool bases the search on user interaction with the solver, during the solution process, as demonstrated at the workshop. This requires manual intervention, where the ex nihilo operator manually decreases the temperature via a scroll bar, although a simple scripted annealing schedule is also available.

During our demonstration this year, we will be using either the manual technique described above, or a simple scripted annealing schedule. For our August 15 2002 delivery, our scripted annealing schedule will be a simple logarithmic cooling of the system. Our work on automated the annealing schedule based on pattern recognition, entropy, phase change, and other thermodynamic metrics is scheduled towards the end of our proposal (2004).

Polling Time for Ex Nihilo INPUTS: (Seconds, Real*4)

(Superceded by Ex Nihilo HTTP Server)

This is a proposed input, and does not exist in the current model. To operate in near-real-time, it will be necessary for ex nihilo to periodically check for updates to key inputs to the model. Any of the key inputs could potentially be redefined during the model run. A balance is needed between minimizing interruptions to the solver (low polling rate), and responding to system changes in a timely manner (high polling rate).

In the Aug. 15 2002 version of the model, ex nihilo has been turned into an HTTP server. A request for load balancing from one of the ex nihilo clients provides the needed inputs via an HTTP call, thus making the startup of the solver open to calls over the net.

Clock Time of Required Ex Nihilo OUTPUTS: (Date/Time?)

(Superceded by Ex Nihilo HTTP Server)

This is a proposed input, and does not exist in the current model. In addition to polling for changes to ex nihilo input, the model will need some form of target time, by which Ultralog will expect answers. An alternative, or addition, would be to have the optimizer provide periodically updated estimates of ex nihilo's best estimate of a good solution, and have the Ultralog oracle (described in earlier posts) do the periodic review of current estimates of optimal modes of operation.

In the Aug. 15 2002 version of the model, the time to solution is "soft wired" to 4 minutes and 30 seconds. The ex nihilo clients may retrieve answers from the ex nihilo server at any time after completion of the solver run.

Conclusion:

In the last three sections we covered the entire set of inputs that are required by ex nihilo. There are four metrics used in our objective function:

1.) Response Time for a set of tasks selected by the user of ex nihilo. To clarify our earlier statements about "user-initiated tasks", these tasks can be any tasks selected for tracing response time. The optimizer does not care if these are user-initiated tasks, or subtasks spawned by previous agent calls in the calling tree.

2.) Probability of Failure of tasks selected by the user of ex nihilo.

3.) Fixed Cost of the entire system modeled by ex nihilo.

4.) Monthly Cost of the entire system modeled by ex nihilo.

Other metrics are possible, such as "probability of compromised intelligence", and various reinterpretations of cost metrics. In the next section we will list a summary of ALL ex nihilo inputs, which is a review of the last three sections.

10. Summary of Input to Ex Nihilo

In the previous three sections, we discussed the entire set of inputs required for operation of ex nihilo. Since some of these inputs had considerable discussion, we are going to condense the entire input set in this section, to give the reader a quick overview of data requirements for the model. For further details, please refer to the following sections:

Chapters with Detailed Descriptions of Ex Nihilo Inputs:

7.0 Function Definition, Performance, Turing Halting Problem

8.0 Host and Data Link Definition

9.0 Miscellaneous Run Parameters, Background Modeling

There are six major sections in our summary of required inputs to ex nihilo:

1.) Function Definition

2.) Host Definition

3.) Link Definition

4.) Miscellaneous Run Parameters

5.) Proposed NEW INPUTS to ex nihilo

6.) Background Loads

We will paraphrase the detailed descriptions in previous posts, and give a short summary of the input set to the optimizer.

FUNCTION DEFINITION:

The ex nihilo tool was designed to model systems of special user-defined functions, and how they call each other and place loads on a distributed system. A function may burn CPU on a host, or call another set of subfunctions, at specified calling rates. The "where are the hosts" question is the assignment problem, treated elsewhere. The function definition part of the problem is frequently the most demanding part of the overall problem definition.

Performance Modeling Tips:

In performance modeling, it is common to condense large collections of similar jobs into a single job. Another common practice is to group resources by host. If a thread requires a collection of hosts to burn CPU, or send messages to other hosts, then it may be useful to aggregate major portions of the problem, and estimate resource use for a major collection of jobs/subjobs on the same host. These "aggregated jobs" may then be used to define an ex nihilo function. It may be useful to keep the following graphic in mind when generating a collection of functions:

Calculate (burn Host CPU) => Messages (burn bandwidth) => Calculate ...

F1 **=>** ***F2*** **(30 % of F1 calling rate)**

=> ***F3*** **=>** ***F4*** **=>** ***F5***

=> ***F6*** **=>** ***F7***

=> ***F8***

=> ***F9*** **(277 % of F1 Calling Rate)**

=> **etc...**

The function calls (where "***F1*** **=>** ***F2***" means "***F1*** calls ***F2***"), generate messages on the network. Note that this calling may be probabilistic, and the actual calling may be via the blackboard publish/subscribe mechanism. The messaging is frequently the major component of response time (and failure) in distributed systems (slow faulty WAN's).

This view usually helps in understanding how the system flows in a performance model. You will want to isolate a selected set of functions for detailed study, and use background modeling to approximate the remainder of the CPU and data link loads.

Remember to avoid recursion when defining ex nihilo functions. To model real recursion, you will need to use statistical methods to get estimates of depth of recursion versus resource requirements.

It may be useful to have a collection of cloned functions, like F1a, F1b, F1c, etc. to simplify data input, and minimize confusion.

An agent operating like a daemon process, with idle periods interrupted by periods of activity (like a web server daemon), should be modeled as a background load in combination with a function that captures the CPU burn during the active periods.

During 2002, we are only using current ***rates*** of CPU utilization and inter-agent messaging.

Overview of Function Properties:

For any given function F, we will need to know the average amount of CPU used by F, and a list of "subfunctions" that are called by F. For each subfunction, we will need to know the "calling ratio", or number of calls to the subfunctions for each call to F. Each function can call multiple subfunctions, with different calling ratios.

The main properties of these "user-defined functions" are listed below. Throughout this summary of ex nihilo input data, we will list an input type (title of input data), followed by a statement of data format (e.g., Real*4 for single precision real), followed by a text summary of the data element.

Functions in Objective Function Task Set:

(*Set of functions being traced, required input*)

The user of ex nihilo must select a set of tasks/functions to be included in the "objective function" used by the optimizer. For each of these user-selected tasks, we will include terms in the objective function to represent Response Time, Probability of Failure, Fixed Cost, and Monthly Cost. Other metrics could be included, but these four metrics are popular in distributed design.

We will refer to this set of user-selected tasks as the ***Objective Function Task Set***. It is the set of tasks that generate the objective function. Each task/function in the Objective Function Task Set is traced through all subtasks (multiple threads allowed), and evaluated in terms of Response Time, Failure, Fixed Cost, or Monthly Cost. The response time of a single user-selected task is determined by the longest thread in the set of threads generated from the user-selected task.

We will occasionally refer to the user-selected tasks as the "initial tasks", and similar terminology, since these tasks are the source of all threads traced for objective function contributions.

The model does not currently allow asynchronous calls, although that modification is already in our proposed SOW. The asynchronous modifications to the solver will allow functions to spawn other processes that are NOT traced for response time. The asynchronous processes would be tracked through all calls to a halting state, to properly account for load levels.

For the year 2002, we are only tracking probability of failure in the objective function. Therefore, ALL functions in the running agent system are listed in the Objective Function Task Set. In the years 2003-2004, we will be tracking multiple classes of users (agents), which requires more a proper subset of all running agents.

Call Rate for Functions in Objective Function Task Set:

(Calling Rate for "Traced" Tasks in Objective Task Set, Calls/Sec, Real*4)

(Only required for response time, 2003-2004)

The user of ex nihilo must specify a calling rate (calls/sec) for the functions in the Objective Function Task Set. These calling rates, when combined with subfunction calls, determine the overall loads placed on the servers and network links (CPU/bandwidth).

The calling rates in models are frequently based on rates of expected job submission to the system, from real human users, or expected rates of "agent calls". The calling rates for this initial set of tasks may also be based on average rates expected from earlier points in the calling sequence. For example, a set of other agents may require a subtask to be performed. In this case, the subtask is like a "subroutine" that may be common to multiple agents, and would be a good candidate for a user-defined function in ex nihilo. To properly define a function in the Objective Function Task Set, the user must specify the rate of calling the function, and other function properties like total resource usage (total CPU), rates of resource usage (CPU sec per sec), and calls to other subfunctions.

Subfunction Calling Ratio:

(0-9.e19, Real*4)

For each subfunction ***Fx*** called by a function ***F***, we need the percentage of subfunction calls to ***Fx*** for each call to ***F***. This is simply the ratio of call rates for the called function and calling function. This field is also useful for modeling functions that make probabilistic calls to other functions.

For example, if a function ***F*** makes a call to another function ***Fx*** 50% of the time, then the calling ratio is 0.5. If each call to ***F*** results in three calls to ***Fx***, then the calling ratio is 3.0.

Function Call Message Size:

(Bytes, Real*4)

For each function call from ***F*** to a subfunction ***Fx***, we need the size of the message sent from ***F*** to ***Fx***, in units of Kilobytes.

Function Eligibility:

(Integer: 0,1)

For each function ***F***, we need the set of hosts on which the function ***F*** may run. Note that eligibility does not imply that the solver will actually choose to execute the function on a given eligible server. A function ***Fb*** called FROM a server ***Sa*** will be assigned to execute on a server ***Sb*** based on the current state matrix.

Although the server assignment for calls FROM a particular host are defined to be on one and only one server, this does not preclude the called function from running on other servers when called FROM other hosts.

Function CPU Burn:

(Seconds, Real*4)

For each function ***F***, and for each server ***S*** for which the function is eligible for execution, we need the amount of CPU burned during the execution of the function ***F*** on the server ***S***.

Function Fixed Cost:

(Dollars, Real*4)

For each function ***F***, and for each eligible server ***S***, we need the fixed cost of installing the function ***F*** on the server ***S***. Cost is open to broad interpretation.

Function Monthly Cost:

(Dollars, Real*4)

For each function F, and for each eligible server S, we need the monthly cost of running the function F on the server S.

HOST DEFINITION:

Host MAIN Properties:

Type:

Server, Shared LAN Hub, Router, Virtual Link Node

A host in an ex nihilo design is actually just a connection point in a network, and may be one of three broad categories of hardware devices.

If the object has a "Server" Type, then it is allowed to send and receive messages, and compute functions (use CPU and memory), but is not allowed to forward message packets to other computers.

If the object is a "Shared LAN Hub" type, then the object is a shared LAN hub, and has the ability to broadcast messages to all other hosts connected to the hub, in standard shared hub fashion.

If the object is a "Router" type, then it is a router, and has the ability to forward messages. A Switched Ethernet can be modeled with a router object and directed Link objects.

Name:

40 Character Text String

The Host Name property is just the text name of the host object. For example, a host associated with processing paychecks might be labeled "Host-paycheck". Any ASCII characters, including spaces, are allowed in the Name property.

Number:

Integer, 1 - numhost

Each host in the network has a unique positive integer, referred to as the Host number. If there are "*numhost*" hosts in the current design, then the number must be between 1 and *numhost*. All IO, including flat file input and output, is listed in order of Host Number.

Host CONFIGURATION Properties:

Background CPU Utilization:

*Percent, Real*4*

This is an estimate of the background load on the server. For the Ultralog "Send OPlan" example, it should be a measure of percent CPU utilization after the app-servers and nodes have been launched, but before the "Send OPlan button" is pressed. In this case, we have an estimate of the background load of the system, before processing starts.

Max CPU Utilization:

*Percent, Real*4*

The Max CPU Utilization property is a user-definable constraint on CPU utilization. For example, a common limit on CPU utilization is 80% or 90%, to give a safety margin to the server. Above 80-90%, it is common for queues to start growing quickly in multi-server queuing systems.

Number of CPU's:

Integer

Each Host of the Server Type must have a defined number of CPU processors greater than zero. This field is ignored for Routers and Shared LAN hubs. For Server types, the number of processors is needed for our CPU queuing model of response time, and for the cost model.

Max Number of CPU's:

Integer

Each Host of the Server Type must have a defined MAXIMUM number of CPU processors greater than zero, which is the Max Number of CPU's property. This field is ignored for Routers and Shared LAN hubs. It is used during the "subspace iteration" part of the search to configure servers to obtain good response time at minimum cost, and is used in systems design mode, as opposed to real-time use in dynamic task assignment (Ultralog).

Memory:

MB, Integer

For each host, we need to know the amount of physical memory present in that host, in units of Megabytes (MB). The Memory property is used as a constraint on the server, and is also used in the cost calculation.

Number of Disks:

Integer

Each host is allowed to have a specified number of hard disk drives. The Number of Disks property is only used in the cost calculation. A disk model enhancement has been discussed elsewhere.

Host COST Properties ($$$):

Installation Cost:

*Dollars, Real*4*

The Installation Cost property is the fixed cost of the host, including installation, in dollars, but *without* memory, CPU, or Hard Disk. It is the price of a raw host, with floppy drive.

Cost per Month:

*Dollars, Real*4*

The Cost per Month is the monthly cost of maintaining the host. It should include regular labor, building lease, server lease, insurance, 24 support, software lease, etc.

Cost per CPU Processor:

*Dollars, Real*4*

The Cost per CPU Processor is the cost of an individual CPU chip.

Cost per Disk:

*Dollars, Real*4*

The Cost per Disk is the cost of an individual hard disk drive.

Cost per MB Memory:

*Dollars, Real*4*

The Cost per MB Memory is the cost of one megabyte of memory.

Host MISCELLANEOUS Properties:

Message Forwarding:

Integer: 0,1

The Message Forwarding property determines whether the host is allowed to forward IP packets in a network environment. If the value of message forwarding is set equal to "1", then IP forwarding is allowed. If the value is set equal to "0", then IP forwarding is not allowed. Always set packet forwarding to "1" for Routers and LAN hubs. For servers, set the value to "0".

Failure Probability:

*Real*4: 0.0 - 1.0*

The Failure Probability property is the probability of failure of the host, in some (unspecified) user measurable time T. For example, 0.001 is a common failure rate for commercially advertised hosting services (non critical service), and is usually quoted in service descriptions as "99.98% uptime".

Max Packets/Sec:

*Real*4, NOT MODELED*

The Max Packets/Sec property is currently not used, but does show up in the GUI and underlying data structures. It is primarily used as a constraint on routers and network cards.

Host GUI Properties:

Color of Host:

Color of Host Text:

Host Position on 640x480 screen:

These are fairly obvious properties, and only used in the ex nihilo GUI. For our demo this year, we will just pick some nice colors and positions for the servers.

LINK DEFINITION:

Link MAIN Properties:

Link Type:

Shared Hub Link or Directed Link

A link in an ex nihilo design is a data link connecting two defined Hosts in the network. The link type must be either a Shared Hub Link, or a Directed Link.

A Shared Hub Link type is only used to model data links connected to shared hub based LANs. This may also be used for shared data links in a "daisy chain" type of LAN topology. In modeling information and traffic in a shared hub environment, the traffic is shared with all hosts on the LAN. The messages are broadcast, in typical shared hub fashion, so a message from one host on the hub to another host on the hub is actually broadcast to ALL of the hosts on the hub. The hub based links are mathematically bi-directional, although only a single link in the GUI model is used to model bi-directional traffic (both directions have same bandwidth and traffic). By convention, a Shared Hub Link in the ex nihilo GUI must originate at the server, and terminate at the hub. This connectivity order is used in the model to simplify shared hub definition, and minimize the number of links used to define a shared hub network. The ex nihilo data check utility will warn the user if a link is connected to hub in the wrong direction (physical direction from hub to server, as opposed to the required server to hub connectivity).

A Directed Link is the other basic Link Type. Links that have the Type property set to "Directed Link" are directional. A switched Ethernet hub can be modeled as a collection of Directed Links connected to a Router Type of Host object.

Link Name:

30 Character Text String

The Link Name property is just the text name of the data link object. For example, a link connecting server S1 to server S2 might be named "S1->S2". Any ASCII characters, including spaces, are allowed in the Name property.

Link Starting Node:

Integer, 1-numhost

The Starting Node is the Host Number at the start of a Directed Link. For Shared Hub Links, this must always be the Host Number of the Host connected to the Hub.

Link Ending Node:

Integer, 1-numhost

The Ending Node is the Host Number at the end of a Directed Link. For Shared Hub Links, this must always be the Host Number of the Hub.

Link CONFIGURATION Properties:

Link Latency:

*Seconds, Real*4*

The Link Latency property is the standard one-way latency of the data link, measured in seconds.

Link Bandwidth:

*Kbit/Sec, Real*4*

For directional links, the Link Bandwidth property is the one-way bandwidth of the data link. For Shared Hub links, this is the bandwidth rating of the Hub object (assuming the bandwidth ratings of the cabling and network cards supports the hub bandwidth rating).

Link Background Load:

*Percent Bandwidth, Real*4*

The Link Background Load property is the percentage of link bandwidth used by background processes not explicitly modeled in ex nihilo.

Probability of Failure:

*Real*4: [0.0 - 1.0]*

The Probability of Failure property is the probability of failure of the Link, in some (unspecified) user measurable time T.

Protocol Overhead:

Bytes, Integer

For a given TCP/IP protocol, this is the number of Bytes of Overhead information associated with a given packet. The Overhead section of the packet contains information like IP addresses, "Path MTU Discovery Flag", and other information not directly associated with the actual user data in the packet.

Maximum Transmission Unit (MTU):

*Bytes, Integer (Real*4 internally in code)*

The Maximum Transmission Unit (MTU) is the maximum size of the data area at the network layer of the TCP/IP protocol stack. For packets with data sizes exceeding MTU, then the model will fragment the packets into multiple smaller packets to meet the MTU constraint. The sum of MTU plus Protocol Overhead is equal to the physical size of the packet or datagram.

Link COST Properties:

Link Installation Cost:

*Dollars, Real*4*

The Link Installation Cost property is the fixed cost of the Link, including installation, in dollars.

Link Cost Per Month:

*Dollars, Real*4*

The Link Cost per Month is the monthly cost of maintaining the data link. It should include regular labor, monthly lease, 24 support, etc.

Link GUI Properties:

Link Line Color:

Link Line Width:

Save Link Icon Position:

These are fairly obvious properties, and only used in the ex nihilo GUI. For our demo this year, we will just pick some nice colors and positions for the links.

Miscellaneous Run Parameters:

In addition to defining Hosts, Data Links, and Functions used in an ex nihilo run, the model also requires a small set of Miscellaneous Run Parameters to fully specify the problem. These run parameters are used to define the objective function, specify initial starting solutions, times for input/output, solver options, and a few GUI parameters.

Weighting of User-Initiated Functions:

*Real*4*

To track and optimize multiple agent tasks in the Objective Function Task Set, there needs to be weighting factors to weight the relative importance of the tasks. For each task in the traced task set, a real number weight must be assigned to the task. The relative importance of each task's contribution to the overall objective function is given by the ratio of the task's weight to other task weights being traced for objective function contributions. For example, consider two tasks, labeled TASK1 and TASK2, that are being traced with weights ***W1*** and ***W2***. Let the objective function contributions be labeled as ***OBJ1*** and ***OBJ2***, and let the overall objective function be labeled as ***OBJTOT***. Then the objective function is given by the following expression:

OBJTOT = W1 * OBJ1 + W2 * OBJ2

The problem of choosing relative weights is a difficult, and ill-defined problem, as it is with any objective function used to optimize a system with different goals and objectives. For most cases, the ex nihilo user is more interested in meeting a set of "satisfactory constraints", which are well defined. The solver will search for operational states that first satisfy the constraint set, and secondarily optimize performance based on the relative weights defined above.

A suggestion for weighting would be to first define constraints on response time, etc., for each agent task of interest, and then assign relative user-defined weights that are roughly equal, in the range of 1.0 to 10.0. A first estimate of weighting might be to scale the weights by 10%, so a simple three agent system might have user-defined weights like 1.0, 1.1, and 1.2. In this example, the solver would give a mild preference to the third agent for optimization, with a weighting of 1.2. Assigning large user-defined weight differences, like 1.0, 100.0, 10000.0, although well defined, and capable of being handled by the solver, might heavily skew the solver to favor the largest weighted agent, and ignore the other agents.

Since agents may have radically different contributions to the objective function, then ex nihilo will generate a "dynamic objective function" to be used to simplify user input.

An ex nihilo user can approximate equally weighted agents by setting all weights equal (e.g., W1=W2=W3...=1.0). The solver will automatically create a dynamic objective function to produce a reasonable estimate of equal weighting.

Weighting of Objective Function Components:

*Real*4*

Since the tool tracks user-defined functions for response time, failure, fixed cost, and monthly cost for a given task, there needs to be some form of prioritizing or weighting of the individual components that make up the objective function. The problem of comparing cost (in dollars) to response time (in seconds) is at best ill-defined. As with the problem above (Weighting of User-Initiated Functions), the constraints in the problem are probably the easiest way for a user to find good solutions to the general problem of optimizing with respect to conflicting goals, in different units.

Consider a single task, with an objective function contribution ***OBJ1*** (e.g., a single agent case, as described above). The optimizer calculates ***OBJ1*** based on the response time, failure, fixed cost, and monthly cost for this task as follows:

OBJ1 = Wt * Rt * T + Wp * Rp * P + Wfc * Rfc * Cfc + Wmc * Rmc * Cmc

where:

Wt = user defined weight for response time

Wp = user defined weight for failure probability

Wfc = user defined weight for fixed cost

Wmc = user defined weight for monthly cost

Rt = Automatically generated Response Time scaling factor (internal to ex nihilo)

Rp = Automatically generated Failure scaling factor (internal to ex nihilo)

Rfc = Automatically generated Fixed Cost scaling factor (internal to ex nihilo)

Rmc = Automatically generated Monthly Cost scaling factor (internal to ex nihilo)

T = Response Time of task being tracked

P = Probability of Failure of task being tracked

Cfc = Fixed Cost of task being tracked

Cmc = Monthly Cost of task being tracked

The automatically generated internal weights, ***Rt***, ***Rp***, ***Rfc***, and ***Rmc*** are described in detail in the section titled "9. Miscellaneous Run Parameters, Background Modeling".

The ex nihilo user is responsible for defining the weights ***Wt***, ***Wp***, ***Wfc***, and ***Wmc***.

Using the dynamic objective approach, the user now only needs to set ***Wt***=***Wp***=***Wfc***=***Wmc*** to define a problem where we have approximately equal weighting of Response Time, Probability of Failure, Fixed Cost, and Monthly Cost.

In light of the above, the main requirement from the user for weighting factors for objective function components is a set of user-defined weights for each component. The values of the weights are best kept in a range of values near 1.0 to 10.0 to produce reasonable results reflecting user preferences.

Initial Starting Solution:

At the start of the solution process, ex nihilo requires an initial starting solution, or seed solution, to begin the search. It helps if the starting solution is a "good solution", but it is not required. The starting solution is not even required to be a physically reasonable solution, and may violate all of the user-defined constraints for response time, failure, and cost. As long as it represents a well defined state of the system, the solver will be able to search the solution space for near-optimal solutions. If the starting solution is a good solution, then it will take less time to converge to a near optimal solution, so it is to the user's advantage to choose a good starting solution if one is available.

There are two possible ways to use a starting solution. In the first method, the user selects a totally random state. This is the method I demonstrated at our workshop, and is the required method when there are no previously stored "good solutions" available to the user. The second method is to use a solution that has been previously generated and stored by the solver. The program has a utility for saving and restoring valid states of the system. Any of the states saved by this utility are valid solutions to read into the solver as the Initial Starting Solution. During dynamic real-time use, the starting solution may be the current operational state of the system.

Min and Max constraints on Response time, Failure, Cost, and Monthly Cost:

*Real*4*

The details and discussion of the constraints on objective function components have been presented in the second section in the series, entitled "2. Thrashing, Satisficing". The constraints have the same meaning for all four components of the objective function.

The Max constraint is the dividing line between acceptable and unacceptable solutions. For component values less than the Max limit, the solution is acceptable. For values greater than the Max limit, the solution is unacceptable.

The Min constraint is the dividing line between the set of "equally good" solutions at or below the lower limit, and solutions which have variations in goodness. For components less than the Min constraint (e.g., response time < TMIN), then the solver does not attempt to improve the quality of the solution along that dimension of the solution space. For components greater than the Min constraint, the solver will continue to search for better solutions.

As an example, consider the response time limit of TMIN = 0.1 seconds. In this case, the solver will consider a response time of zero (0.0) as equal in goodness to a response time of 0.1 seconds, so it won't waste effort looking for solutions that are "too good" (can't take advantage of further improvements). For solutions with response times greater than 0.1 seconds, then the solver will attempt to improve the solution.

Subspace Iteration:

Check Box

The Subspace Iteration part of the solver is used in "design mode", to find server configurations that satisfy response time constraints at minimum cost. During a subspace iteration, the solver will vary the number of CPU processors defined on each host, and choose an optimal server configuration with respect to CPU count.

For run-time use in optimizing an existing system with well defined hardware (like Ultralog), the Subspace Iteration check box should be left unchecked.

Cost/Failure Routing Corrections:

check box

The "Cost/Failure Routing Corrections" part of the solver is used during run-time to generate an optimal set of routing matrices that are also optimal with respect to Cost and Failure. Recall that normal routing decisions are based on the number of router hops between communicating hosts, which is a much simpler problem than the problem of generating a set of router matrices that minimize Response Time, Failure, Fixed Cost, and Monthly Cost.

For the demonstrations this year, it is recommended that the ex nihilo user turn off "Cost/Failure Routing Corrections" (unchecked). Since we are starting with a fixed logical topology (this year), there is no need to add corrections to the search based on routes optimized for Cost or Failure. Turning off the Cost/Failure Routing Corrections will speed up the ex nihilo solver.

Screen Refresh Rate:

*Seconds, Real*4*

The Screen Refresh Rate is a GUI feature. It is the time that elapses between refreshes of the of the main graphics screen of the ex nihilo solver. It has no impact on the solution, although a high refresh rate will slow down the solver progress due to frequent GUI interrupts.

State Viewing:

GUI, Radio Button

The State Viewing option is a GUI feature. During the solution process, the solver hops around from one state to another state. The annealing state is the current estimate of the "best solution", as defined by the annealing search. Since the solver may hop out of the true best state found in a search, then the annealing state may or may not be the actual best state found in the search. It can be shown that as the solution progresses, then the annealing solution will converge in distribution to the global optimum. Since real solutions are usually terminated before the global optimum is found, then the current annealing state may actually be worse than previously discovered states. To account for this, the solver stores both the annealing state, and the first three "best states" during the solution process.

The GUI allows the user to choose between viewing the annealing state, or the current state of the system during the search. The current state of the system is mainly used when using the ex nihilo Trace Utility. The Trace Utility provides a complete view of any current state, and includes a function traceback utility, and volumes of output on events associated with that current state.

During solution iteration, leave the State Viewing option on the Annealing/Run page set to "Annealing".

Annealing Temperature:

*Real*4, Scroll Bar Input*

The Annealing Temperature is used by the ex nihilo solver as part of the Simulated Annealing algorithm. A discussion of how simulated annealing is used within ex nihilo is left for future documentation. Although the topic can be extremely theoretical, a quick overview follows:

At high temperature in an optimization problem, the system is allowed to hop around randomly, from one state to the next state, with no preference given to states of lower energy (objective function value). At high temperature, all state transitions are equally acceptable, so the algorithm behaves like a random search. As we lower the temperature, the transition to states with lower energy will be accepted with higher probability (as determined by an algorithm known as the Metropolis algorithm). At very low temperature, the algorithm only accepts transitions to lower energy states, and only moves "downhill". This mix of "random" search at high temperature, and "pure downhill moves" at lower temperature, allows the search technique to move across the hills and valleys of the objective function, and avoid getting trapped in local minima (or maxima). This is what makes simulated annealing a good choice for global optimization.

For ex nihilo to operate properly, it requires a procedure for lowering the annealing temperature from high temperature at the start of the search, to low temperature at the end of the search. The current tool bases the search on user interaction with the solver, during the solution process, as demonstrated at the workshop. This requires manual intervention, where the ex nihilo operator manually decreases the temperature via a scroll bar, although a simple scripted annealing schedule is also available.

During our demonstration this year, we will be using either the manual technique described above, or a simple scripted annealing schedule. For our August 15 2002 delivery, our scripted annealing schedule will be a simple logarithmic cooling of the system. Our work on automated the annealing schedule based on pattern recognition, entropy, phase change, and other thermodynamic metrics is scheduled towards the end of our proposal (2004).

Proposed NEW INPUTS to Ex Nihilo:

In this section we will cover some of the new inputs that have been proposed for use in the distributed agent problem. We are especially interested in comments on the proposed inputs.

Function Call Wait Time:

*Seconds, Real*4*

(Not Implemented, Request for Comments)

This is a proposed input, and does not exist in the current model. This property is a proposed function property. It is intended to help model the blackboard contributions to response time.

Current Tool: (No Wait Times):

The current model was defined to model situations where there are no "idle, delay or wait periods" between the end of execution of one function, and the calls to other subfunctions. This works well in situations where subfunctions are called directly by the function. For example, in our 3-tier web demo at the workshop, we had a web server that called an application server, which called a data server. In this example, there was no delay time between end of web server CPU burn, and the beginning of the messaging event from the web server to the application server (which initiates app server CPU burn).

When calling a subfunction in the current solver, the model updates the "thread clock" at the end of execution of the calling function, and then immediately sends a message to the called function. Thread clock advances during messaging, and continues on the newly called function/server. This means that there are no idle periods in the thread trace to account for periods where real clock time advances without CPU burn or queuing on a server.

Blackboard Wait Times:

In the blackboard problem, a task is posted to a blackboard, and begins execution by a blackboard subscriber at a later time (after the publish event). In this case, there is a real clock delay between published task and assigned task. During this expected blackboard delay time, although the thread clock advances, there is no CPU burn by the function (only background CPU burn).

This modification would place a user-defined delay/wait/pause time into the thread clock AFTER completion of the calling function, and BEFORE a message is sent to the called function. Note that this does not account for polling by the blackboard subscribers, or blackboard itself, which are best handled as background loads. The wait time should be an average expected wait time between the time of a published task, and the time that a task on the blackboard has been assigned to another agent.

To handle this problem at a reasonable level of generality and flexibility, we need to assign wait times for called functions based on both the calling function and calling node. To illustrate the definition of wait time, we will use an example.

Consider the case where an agent F on server S posts a task to the blackboard on server S. In this case, we do not know which of the subscribed agents will handle the task posted to the blackboard. Suppose there are two agents F1 and F2, and that we expect F1 to handle the task request 75% of the time, and F2 to handle the task request 25% of the time. This could be modeled by defining the subfunction calling ratios (defined above in function definition section) in the definition of F. We would say that the calling ratio for F1 is 0.75, and the calling ratio for F2 is 0.25. Suppose further that on average, it takes T seconds from the time F posts its task to the blackboard, until the time that one of the subscribed agents is assigned the task (either F1 or F2). In this case, we would say that the "Function Call Wait Time" for function F calling F1 or F2 is T, when the calling function F is located on server S.

In summary, the Function Call Wait Time property is a three dimensional matrix. For each Server S executing a function F making calls to subfunction Fsub, the following data are required input to the solver:

Wait = ***Wait***(Calling Function ***F***, Called Function ***Fsub***, Calling Server ***S***)

Wait = ***Wait***(***F***, ***Fsub***, ***S***)

(Remember that Asynchronous Function Calls are scheduled later in our Statement of Work.)

11. Simulated Annealing:

The current approaches in Simulated Annealing draw has its history in a field of physics known as "Statistical Quantum Mechanics". The field is a blend of everybody's favorite physics topics: Thermodynamics, and Quantum Mechanics, although it is fairly easy to explain at an overview level.

In combinatorial optimization and distributed design, we have come to a point where we are trying to model dynamic systems of agents, as a molten metal starting to crystallize, or a gas of computer jobs condensing to liquid. There are dramatic points in large complex behaviour, like crystallization, or melting, or vaporizing, that appear to map well to problems in complexity in combinatorial optimization and other fields. In physics, an example of a phase change would be the melting of ice, or crystallization of a silicon chip. In combinatorial optimization, we look for phase changes in the solution space, and use this to help automate our solution process. In distributed agent assignment problems, we are looking for phase changes based on varying degrees of program architecture.

Once we have a basic workload model, and chosen a method for modeling the various components and interactions in the system, we are ready to design an algorithm for searching the design space. As we can see, our search must take place in a space with many local minima, and numerous constraints. Furthermore, the discrete part of the search will be NP-Complete or worse. One of the more popular methods for searching these types of spaces is through the Simulated Annealing algorithm. In simulated annealing, the search proceeds from one “state” to the next state, with a specified probability. A state, in our systems design model, is a fully specified design of a complete system. By “fully specified design”, we mean that the design completely defines the nodes, connectivity, bandwidth, background loads, response times, task assignments, data assignments, messaging, costs, installed software, etc. :

State ≡ State(hardware, software, routing, workload, task assignments,...)

The state transitions form an inhomogeneous Markov chain, where the transition probabilities are dependent on the two states involved in the transition, and a “temperature parameter”, which is discussed later:

$State_1 ==> State_2 ==> State_3 .... ==> State_N$

For each fully specified design state, we define a cost/objective function:

“Cost” = Function of response time, failure, costs, etc.

In the simulated annealing search, if a proposed state transition (or state move) results in an improved state, or "downhill" move in our objective function (e.g., good response time), then the move is accepted. If the proposed transition results in a degraded state, or "uphill" move (e.g., poor response time), then the move is accepted with a probability that is exponentially damped with respect to the cost of the uphill move:

$$P(\text{Accept Downhill Move}) = 1.0$$

$$P(\text{Accept Uphill Move}) = \exp(-\Delta\text{Cost} / T)$$

where T is the temperature used in the simulated annealing algorithm.

The temperature can be loosely thought of as a measure of randomness of the algorithm. At high temperature, almost all proposed transitions are accepted, giving a feel of a purely random search. As the temperature approaches zero, the probability of uphill moves is exponentially damped, so the moves are predominantly downhill. By allowing probabilistic "uphill moves" in the search, the annealing technique avoids getting trapped in the local minima that plague other search techniques [4]. The simulated annealing algorithm is now widely used on a variety of NP-Hard optimization problems.

The annealing search begins with the search at a temperature which is "high" compared to typical changes in the objective function. The temperature is slowly lowered, letting the system reach "thermodynamic equilibrium" at each temperature point. This means that at any given temperature, we sample a large number of states at that temperature before lowering the temperature. Determining the rate at which the temperature is lowered is an active area of research for the algorithm, with recent research in the areas of phase change. In practice, the search is usually performed in several passes though the space, looking for typical changes in the objective function, and for regions in the temperature parameter where the cooling rate should be increased or decreased.

At higher temperature, the algorithm feels out the macroscopic features of the space, and is insensitive to the finer fluctuations in the objective function. As we lower the temperature, the search will focus on the fine grain structure of the space. Under fairly general conditions, the algorithm can be shown to converge in distribution, and probability, to the global minimum.[4]

12. Changes After Albuquerque Meeting, April 2002

The Albuquerque meeting was a watershed event in the application of ex nihilo to the distributed agent problem. The demanding set of inputs required for response time modeling was a problem, since many of the inputs could not be generated for the Aug. 15 2002 deliverable. Although the inputs could be "stipulated", this would not be useful for the demonstration of capabilities associated with a real agent problem.

It is worth emphasizing (due to many questions on the matter), that the *demanding* input requirements have nothing to do with simulated annealing. The few simulated annealing inputs required to control the solver can be easily generated. By contrast, *any* attempt to model response time would require a detailed knowledge of the "agent calling structure", and specifically, the average CPU burn rate of each agent in the system, and messaging rate between agents. Furthermore, the CPU burn rate was the total CPU burn for an agent task, in units of CPU seconds, as opposed to instantaneous rates of CPU usage, in units of CPU seconds per second. In a similar manner, the inter-agent messaging was to be described in units of bytes per message, as opposed to the instantaneous messaging rates in units of bytes per second.

The need for absolute CPU and inter-agent messaging was a problem, since the system at the time was only providing rates of CPU and messaging. There was no work-around for this problem, so it was necessary to postpone the response time modeling until 2003-2004.

To demonstrate the solver for the Aug. 15 2002 delivery, it was necessary to remove the response time modeling from the solver, and focus exclusively on the probability of failure of the agent system. The problem statement was modified for 2002, and restated as follows:

> Adjust the running agent system, so that the solver produces a set of agent assignments that place all physical resources (hosts and links) within the "healthy part of the operating range" of each of the system component. This system produced by the solver should minimize the "probability of hard failure of the running system".

This statement needs clarification in two areas. We need to define the phrase "healthy part of the operating range" of each of the system component, and the phrase "probability of hard failure of the running system"

Healthy Operating Ranges of System Components

The "healthy operating range" must be defined for each node and data link in the system. In both cases, we will define a maximum usable level loading of a component beyond which the component exhibits degraded performance.

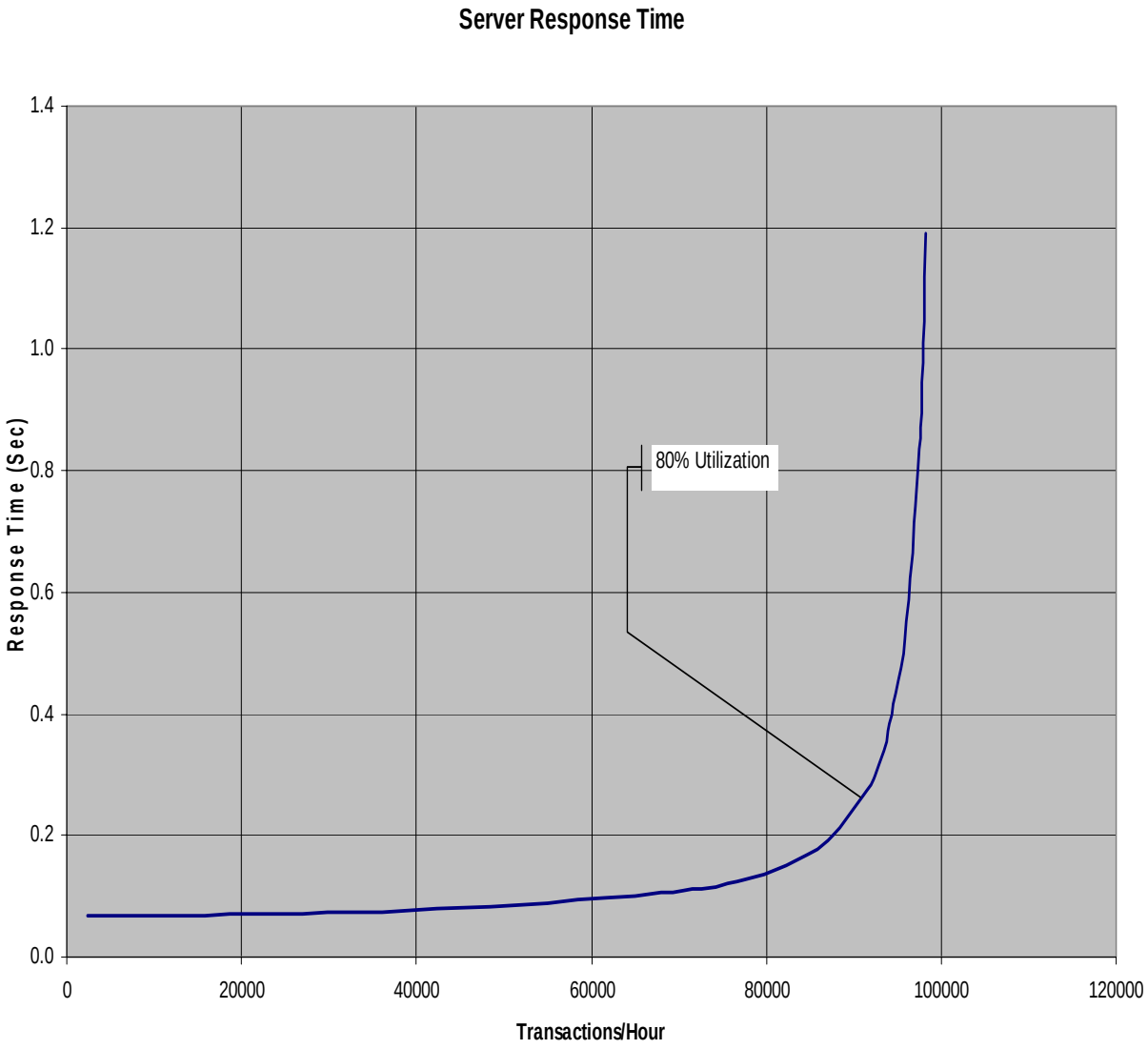
Hosts:

For hosts, we will define health in terms of queuing, paging, and other aspects of performance. In the current model, our primary "resource users" are the agents, operating system, and other loads associated with a running agent system. At present, the current resources we are concerned with are CPU and memory.

Host CPU:

The primary measure of CPU usage is percent CPU utilization. For real processors, the CPU utilization cannot exceed 100%, when the processor is 100% busy. This is a physical limitation on the processor, and therefore places a constraint on the host. However, a processor running at 100% is by no means "healthy". At this level of utilization, we will find that there are tasks that will build up in the processing queue due to random arrivals of tasks at the processor. Alternatively, at this high level of utilization, we may also find that the processor begins time slicing and switching between tasks in a shared processing mode. Each time the system switches between shared tasks, the system undergoes a "context switch". Context switching can be extremely inefficient, and excessive context switching is a sign of poor systems health, just like excessive queue lengths.

The following graph illustrates the effects of queuing at the CPU, from an M/M/n queuing model of transactions submitted to a CPU processor. This is a classic queuing profile, and in this example, shows that the 80% CPU utilization level is a practical limit for this server.



Notice how quickly the system performance declines (response time) above the 80% utilization level in the above example. In this example, the 80% utilization mark is a good estimate of the maximum allowable rate of service in the system component. For this reason, this "knee of the curve of response time" is a good candidate for maximum usable resource utilization. Since this "Max CPU utilization" is a user input, then this is a source of future automation.

In practice, the level of onset of excessive queuing occurs at approximately 80-90% utilization. This is an effective value to use for the "max CPU utilization" in the project input. It is usually helpful in global optimization problems (with notable exceptions), to limit the size of the space being searched. We press the system into the highly nonlinear queuing phase when we push the servers beyond 80-90% CPU utilization. That makes the "knee of the curve", a useful guideline for max CPU utilization limit.

In the current CPU model, the system will search for a design with the proposed inputs. If no solution is found (i.e. distribute system that solves problem), then the model has reached a "hard constraint", meaning "no solution found". The model does not currently violate a hard constraint, like 80% CPU, or 512 MB memory. An attempt to transfer agents onto a host requiring services beyond the server capabilities invalidates the state change request.

One of the main proposals for future modifications to the solver is that the hard constraints be softened. In the modified solver, we would allow constraint violations, on a penalty basis. For the CPU this would mean excessive queuing and context switching, and instability. For the memory, it would mean paging, and disk IO.

Another enhancement to the tool could search for the "knee of the curve" of response time in the above queuing curve, and automatically find good locations for maximum usable computing power (i.e., automatically find max CPU limits).

Host Memory:

In our memory model, we allow each server to have a specified amount of RAM. Each agent is allowed to consume an unshared amount of RAM. It is assumed that the agent RAM consumption is based on the estimated memory size requirement of an agent without memory paging.

The memory model is a simple sum of unshared memory on the same host of all active agents, and acts as a hard constraint. If a proposed state move results in a host being loaded with agents that require more memory than the host has available, then the state is flagged as a failed state, and the solver tries another move.

A proposed change has been made to this hard constraint, in which partial violations of the memory limit would be allowed. This would allow a situation where the system would adjust to find a state with the lowest overall memory constraint violation.

Links:

During the Albuquerque meeting, we also decided that data link traffic would be modeled at an instantaneous level of use, as opposed to absolute message sizes. The need for absolute message sizes is another data requirement for modeling response time. Without the absolute message size, in bytes, sent by one agent to another agent, then it is impossible to make statements about response time, since we cannot make statements about the time required to send a message. As a result of this requirement, we decided to postpone response time, and the corresponding need for absolute message sizing, until 2003-2004.

13. Probability of Failure, and Mean Rehydration Time

Our first goal after Albuquerque was to design a distributed system of agents that satisfy hard constraints on CPU, Memory, and Bandwidth, and which minimize the probability of failure of the system. The probability of failure is a bit subtle, so needs a bit of explanation.

> We will define a system failure as occurring when any of the agents in the system is either unable to run on its current host, or unable to communicate with any of the other required agents in the system. When a failure occurs, we are forced to move some of our agents to other servers to put the system back into operational status.

After a failure, we may have a varying degree of work perform to move the agents to a new server, but for now, let's just try to calculate the probability of failure of the system, regardless of what happens after the system fails.

Single Enclave Example of Failure

Consider a simple problem where we have a single LAN segment (single enclave), and with 56 Agents and 8 Hosts. This has been a popular test case in the TIC, so let's study it in detail. For simplicity, assume that all link failure probabilities are zero, and the switched hub has a zero failure probability, so we are only interested in host failure. Assume further that all hosts have the same probability of failure Pf = 0.8.

Let us study two possible states of the system. Assume further that it is possible for ALL 56 agents to run on a single host in the system. We will compare the probability of failure of the case when all agents are on a single host, to the case when all agents are spread evenly among the hosts.

CASE I: All Agents on 1 Server:

In this case, the probability of the system being "up" is simply the probability of a single host being up, which is (1-Pf).

Pfail(All agents on 1 server operational) = Pf

CASE II: All Agents distributed on All 8 Servers:

In this case, the system will "fail" if *any* of the servers fail, since they are all required for the system to be fully operational. To find the probability of failure for this case, let's first calculate the probability of success that the system is fully operational. To find the overall probability of success, we need to multiply the probabilities of success for all systems components. Since the probability of success for any one host being operational is (1-Pf) then we find that the overall system has a probability of success given as follows:

Psuccess(All agents on all 8 servers operational) = (1 - Pf)8**

Pfail = 1 - Psuccess

Plugging in some numbers, like Pf = 0.8, we find that

Pfail(All agents on 1 server operational) = 0.2

Pfail(All agents on all 8 servers operational) = 1 - (1 - Pf)8 = .83 ...**

This is an important result, and has been the source of much discussion on ex nihilo results. In this example, we find that the failure rate of moving all agents to one server is 20% failure. By way of contrast, if we try to distribute the agents to all servers, the failure rate is greater than 83%!

The physical meaning is obvious. If we have our agent system exposed to more servers, we also expose it to greater opportunities for failure. This is why the tool appears to minimize the number of hosts used to support a system.

Calculation of Mean Rehydration Time

In this section, we are going to calculate the mean time for rehydration for our given LAN example. We will be interested in using Mean Rehydration Time as an updated statement of optimal agent/host assignment. We calculate this quantity by finding the mean time required to rehydrate a system, following a probabilistic failure in the current agent/host assignment.

Let Tmove1 be the time required to move one agent to a new host after a failure. Let us further assume that all moves are performed sequentially (bad approximation, since many moves are in parallel). We will define Tmove_total as the time required to move all agents to new hosts from failed hosts.

In our first case, with all agents on one server, when we lose that server, we lose everything, and all agents must be rehydrated elsewhere, so we need to calculate the mean move time following crash of part of the system. For the case where all agents on one specific server, let us refer to that specific server has H0). Let us refer to the remaining set of servers (excluding H0) as Hunused. Let Tmove(*) refer to the move times required to move various sets of agents:

Tmove_total(all agents on one server) = Na * Tmove1

Using this information, we can compute the mean time spent moving agents as follows:

<Tmove_total0(all agents on one server)> = mean move time = Pf * Na * Tmove1

There is a hidden term in the above expression for the case where all servers fail. In this case, there are no servers available for the move, so we need to remove this term, since the time to move is pointless when all servers are down:

Pfail(all agents on one server) = mean move time = P(1 specific server failure) *(P(0 failures out of N-1) + P(1 failure out of N-1) + ... + P(N-1 failures out of N-1)

We need to modify this, and remove P(N-1 failures out of N-1), since this case corresponds to total failure, and should not be incorporated in our calculation of mean rebuilding time for the system:

<Tmove_total0(all agents on one server)> = mean move time =

= P(H0 fails, 0 Hunused failures) * Tmove(0 agents) + P(H0 fails, 1 Hunused failures) * Tmove(agents on 1 server) + ... + P(H0 fails, all servers in Hunused failed)* Tmove(move agents on all servers)

In the above expression, all move times are the same, since all agents are on the same failed host, and no agent moves are required on the failed set Hunused. We can also pull a common factor of P(H0 fails) out of the summation to find:

<Tmove_total0(all agents on one server)> = mean move time =

= Tmove (all agents on H0) * Pf * [P(0 Hunused failures) + P(1 Hunused failure) + ... + P(N-1 Hunused failures] = Na * Pf

Since we are omitting the last term where everything in the system fails (P(N-1 Hunused failures), then proceed as follows, and define Tmove_total as the total estimated mean time for rehydration of a system, given that there is at least one eligible server capable of hosting agents after partial system failure.

<Tmove_total(all agents on one server)> = useful mean move time = <Tmove_total0(all agents on one server)> - P(H0 fails, all servers in Hunused failed)* Tmove(move agents on all servers)

= Na * Tmove1 * Pf - P(H0 fails, all servers in Hunused failed)* Tmove(move agents on all servers)

= Pf * Na * Tmove1 - Pf**Nh * Na * Tmove1

= Pf * Na * Tmove1 * (1 - Pf **(Nh-1))

Therefore *in this example*, our estimate of mean rehydration time associated with system failures, of a single LAN model with perfect links and routers and Na agents Nh Hosts, and equal host failure probabilities, given that any single host is capable of running all agents at once, is given below: For the case Nh = 1, then the Tmove_total is undefined, since LAN's with a single host, after single host failure, is a dead system. The <variable> is used frequently in statistics, as a shorthand notation for "average, or mean expectation".

<***Tmove_total(***all agents on one server)> = ***Pf*** * ***Na*** * ***Tmove1*** * (1 - ***Pf*** **(***Nh*** -1))

Case II: Agents equally distributed

In our second case, all Na agents are equally distributed among the Nh hosts, and that there is a single state manager (solver, outside ex nihilo), that rehydrates hosts sequentially. In this case, we would add all host restoration times in a single thread. We will look at parallel threads later.

Tmove(one server fail) = (Na / Nh) * Tmove1

The probability of a single server failure is P(1 host fail):

P(1 host fail) = C(Nh,1) * Pf * (1-Pf)**(Nh-1)

For hosts distributed among all servers, we also need to compute the restoration times for multiple host failures:

Tmove(2 server failures) = 2 * (Na / Nh) * Tmove1

P(2 host failures) = C(Nh,2) * Pf **2 * (1-Pf)**(Nh-2)

... etc.

Tmove(N server failures) = Nh * (Na / Nh) * Tmove1 (totally failed state)

P(N host failures) = C(Nh,Nh) * Pf**Nh

So the total expected server move times would be:

<Tmove_total(all agents on all servers)> = Sum from 1 to N of Tmove(i failures) * P(i host failures)

= sum {(i * (Na / Nh)* Tmove1) * (C(Nh,i) * Pf**i * (1 - Pf)**(Nh-i))}

= (Na / Nh) * Tmove1 * sum (1 to Nh) {i * C(Nh,i)*Pf**i * (1 - Pf)**(Nh - i)}

= (Na / Nh) * Tmove1 * sum (0 to Nh) {i * C(Nh,i)*Pf**i * (1 - Pf)**(Nh - i)}

(last step since i = 0 in sum has no contribution)

f(x) = (x + y) **N = sum (0 to N) (C(N,i) x**i * y**(N-i))

df/dx = N * (x + y) **(N-1) = sum(0 to N) (C(N,i) * i * x **(i-1) * y**(N-i))

= (1 / x) sum(0 to N) (C(N,i) * i * x**i * y**(N-i))

therefore,

sum(0 to N) (i * C(N,i) x**i * y**(N-i)) = x * N * (x + y) **(N - 1)

so we find the following expression for the mean agent rehydration time, when all servers are used, and with a singly threaded rehydration solver:

<Tmove_total(all agents on all servers)> = (Na / Nh) * Tmove1 * sum (0 to Nh) {i * C(Nh,i)*Pf**i * (1 - Pf)**(Nh - i)}

= (Na / Nh) * Tmove1 * Pf * Nh * (Pf + (1 - Pf)) **(N-1)

= (Na / Nh) * Tmove1 * Pf * Nh

= Na * Tmove1 * Pf

As in the preceding case single service case, we want to extract the term involving rehydration time in cases where all servers have failed.

Pfail(agents on all server) = P(1 specific server failure) *(P(0 failures out of N-1) + P(1 failure out of N-1) + ... + P(N-1 failures out of N-1)

We need to modify this, and remove P(N-1 failures out of N-1), since this case corresponds to total failure, and should not be incorporated in our calculation of mean rebuilding time for the system:

Therefore, we have:

<Tmove_total(all agents on all servers)>

= Na * Tmove1 * Pf - (Tmove1 * Na / Nh) * Nh * C(Nh,Nh)*Pf**Nh * (1 - Pf)**(Nh - 1)

= Pf * Na * Tmove1 * (1 - Pf**Nh * (1 - Pf)**(Nh-1))

Now we want to compare this to the mean time

<***Tmove_total*(**all agents on one server)> = ***Pf*** * ***Na*** * ***Tmove1*** * (1 - ***Pf*** **(***Nh*** -1))

CASE II: Agents on All Servers

<***Tmove_total*(**all agents on all servers**)**> = ***Pf * Na * Tmove1* * (1 - *Pf***Nh * (1 - *Pf*)**(*Nh*-1))**

So the question is to find which is faster, rehydration of all agents on a single server, or rehydration after agents have been assigned equally to all servers?

<***Tmove_total*(**all agents on one server**)**> is less than <***Tmove_total*(**all agents on all servers**)**>

if and only if

Pf * ***Na*** * ***Tmove1*** * (1 - ***Pf*** **(***Nh*** -1)) < ***Pf * Na * Tmove1* * (1 - *Pf***Nh * (1 - *Pf*)**(*Nh*-1))**

(1 - ***Pf*** **(***Nh*** -1)) < **(1 - *Pf***Nh * (1 - *Pf*)**(*Nh*-1))**

- ***Pf*** **(***Nh*** -1) < **- *Pf***Nh * (1 - *Pf*)**(*Nh-1*)**

Pf **(***Nh*** -1) > **Pf**Nh * (1 - *Pf*)**(*Nh*-1)**

Pf **(***Nh*** -1) > **Pf**Nh * (1 - *Pf*)**(*Nh*-1)**

1 > **Pf * (1 - *Pf*)**(*Nh*-1)** **(always true)**

Therefore the mean hydration time is better, in this example, when all agents are on the single server, provided that the rehydration process is singly threaded (one rehydration manager).

So now let's add the possibility of parallelism in the rehydration process. Suppose we divide the processing time for rehydration, which changes our expected rehydration time.

For agents distributed among the hosts, the first estimate assumed a singly threaded solver for rehydration (single agent manager). For hosts distributed among all servers, we also need to compute the restoration times for multiple host failures, but now account for the parallelism in estimating overall node and server rehydration. To calculate the overall "mean system rehydration time", we will need to find the longest thread in the set of hosts undergoing rehydration. At this point, we should point out that this has not been implemented in ex nihilo, although this is similar to the thread tracking performed in ex nihilo, so we know it is possible, within reason on the numbers of hosts, links, routes, pfail...

Let's look at the case where we have 2 server failures, with agents distributed equally among all servers:

Tmove(agents on all servers, 2 server failures, single thread rehydration) = 2 * (Na / Nh) * Tmove1

In the new parallel example, we have

Tmove(agents on all servers, 2 server failures, Nh-threaded rehydration) = (Na / Nh) * Tmove1

basically, we are parallelizing by the number of hosts serving the agents. So the total expected parallel threaded solver manager would have the following expected run time: What we see, is that in our summation over all failure states, that the rehydration time is independent of the number of failed hosts (factor of 2 in the above).

<Tmove_total(all agents on all servers, parallel threaded)> = Sum from 1 to N of Tmove(agents on specific host) * P(i host failures)

= sum (1 to Nh) {((Na / Nh)* Tmove1) * (C(Nh,i) * Pf**i * (1 - Pf)**(Nh-i))}

= (Na / Nh) * Tmove1 * sum (1 to Nh) (C(Nh,i)*Pf**i * (1 - Pf)**(Nh - i))

= (Na / Nh) * Tmove1 * [sum (0 to Nh) {C(Nh,i)*Pf**i * (1 - Pf)**(Nh - i)} - C(Nh,0)*1 * (1 - Pf)**Nh]

= (Na / Nh) * Tmove1 * [1 - (1 - Pf)**Nh]

<Tmove_total(all agents on all servers, parallel threaded)> = Sum from 1 to N of Tmove(i failures) * P(i host failures) = (Na / Nh) * Tmove1

<*Tmove_total*(all agents on all servers, parallel threaded)> = Sum from 1 to N of Tmove(i failures) * P(i host failures) = (*Na* / *Nh*) * *Tmove1* * [1 - (1 - *Pf*)Nh*]**

As in the preceding case where all servers were equally involved, we want to extract the term involving rehydration time in cases where all servers have failed.

Pfail(agents on all server) = P(1 specific server failure) *(P(0 failures out of N-1) + P(1 failure out of N-1) + ... + P(N-1 failures out of N-1)

We need to modify this, and remove P(N-1 failures out of N-1), since this case corresponds to total failure, and should not be incorporated in our calculation of mean rebuilding time for the system:

Therefore, we have:

<Tmove_total(agents on all servers, parallel threaded)>

= Na * Tmove1 * Pf - (Na / Nh) * Tmove1

<*Tmove_total*(agents on all servers, parallel threaded)> = *Na* * *Tmove1* * *Pf* - (*Na* / *Nh*) * *Tmove1*

Now compare multihost problem, with a parallel threaded host rehydration solver, to the case of all agents move to single host.

<*Tmove_total*(all agents on one server, Single threaded rehydration)> = *Pf* * *Na* * *Tmove1* * (1 - *Pf* **(*Nh* -1))

<*Tmove_total*(agents on all servers, parallel threaded rehydration)> = (*Na* / *Nh*) * *Tmove1* * [1 - (1 - *Pf*)Nh*]**

Let us put in some modifications for CASE I, and consider the situation where all agents are assigned to a single host, but with parallel threading on the rehydration solver. The modification is fairly obvious, and is stated below:

$$<\textbf{\textit{Tmove\_total}}(\text{all agents on one server, parallel threaded rehydration})> = \textbf{\textit{Pf}} * (\textbf{\textit{Na}} / \textbf{\textit{Nh}}) * \textbf{\textit{Tmove1}} * (1 - \textbf{\textit{Pf}} ** (\textbf{\textit{Nh}} - 1))$$

When is the multithreaded host distribution better than the single threaded rehydration solver?

<***Tmove_total*(**all agents on one server)> is less than **<*Tmove_total***(agents on all servers, parallel threaded) if and only if following is true:

$$\textbf{\textit{Na}} * \textbf{\textit{Pf}} * \textbf{\textit{Tmove1}} * (1 - \textbf{\textit{Pf}} ** (\textbf{\textit{Nh}} - 1)) < (\textbf{\textit{Na}} / \textbf{\textit{Nh}}) * \textbf{\textit{Tmove1}} * [1 - (1 - \textbf{\textit{Pf}}) ** \textbf{\textit{Nh}}]$$

$$\textbf{\textit{Pf}} * (1 - \textbf{\textit{Pf}} ** (\textbf{\textit{Nh}} - 1)) < (\mathbf{1} / \textbf{\textit{Nh}}) * [1 - (1 - \textbf{\textit{Pf}}) ** \textbf{\textit{Nh}}]$$

Consider the Nh=2 case.

$$pf * (1 - pf) < (1/2) * (1 - (1-pf)**2)$$

$$2 * Pf - 2 * Pf**2 < (1 - (1 - 2 * Pf + Pf**2)) = 2 * Pf - Pf**2$$

$$- 2 * Pf**2 < - Pf**2$$

$$- Pf**2 < 0$$

The last step is always true, so in the two server case with parallel threaded rehydration solvers, we always want to move ALL of our agents to a single host, if we want to minimize the overall rehydration time of the system. Now lets look at the high server limit, so we can approximate Nh with Nh-1. To further define the problem, break it down into two cases, low Pf and high Pf:

Let's find the limit of this as Nh goes to infinity. Define a function f(x), and when f(x) is less than one, then it is better to move all hosts to one server, compared to equal distribution on all servers.

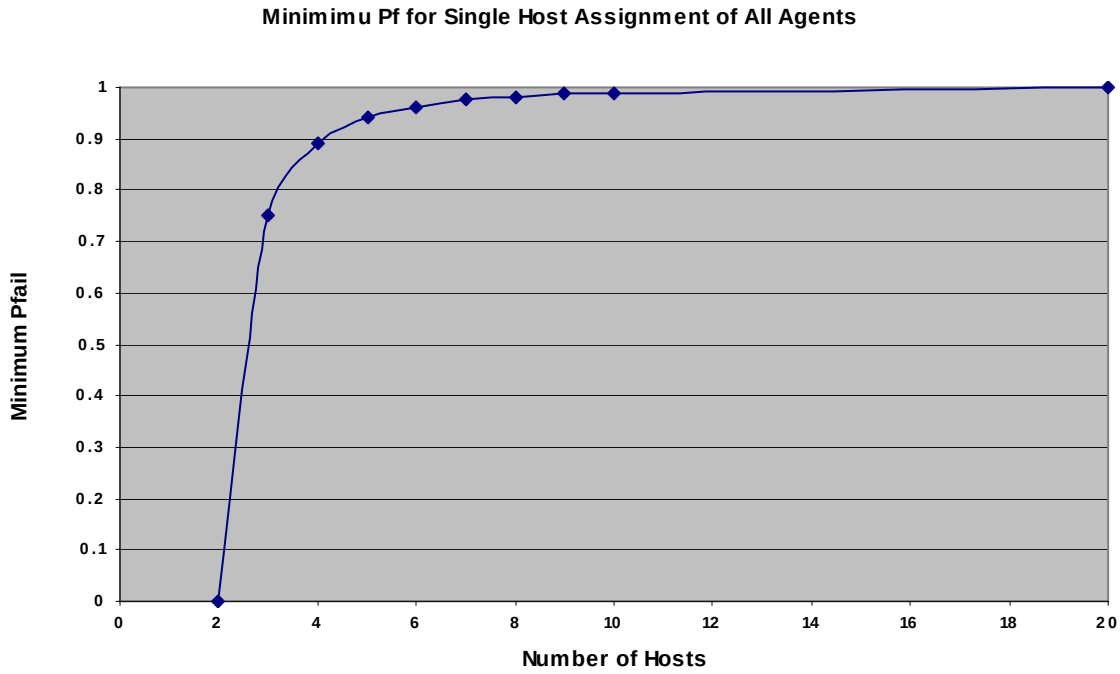
$$f(Nh) = (Nh * Pf) * (1 - Pf**(Nh-1)) / (1 - (1-Pf)**Nh) < 1$$

This is the primary result of this simple analysis, namely, that using the definition of mean rehydration after failure as our primary unit of goodness in a solution state, we find that it is preferable to host all agents on one server, when the following expression is true:

(Nh * Pf) * (1 - Pf(Nh-1)) / (1 - (1-Pf)**Nh) < 1**

The above condition is the condition under which we would move agents to a single server, in a parallel threaded rehydration environment.

Note that we can graph the function f(Nh, Pf) for fixed Nh, and get an estimate of the range of probability for which hosting all agents on a single host is preferred over hosting all agents on ALL of the servers. Note also, that there are multiple subcases involving hosting all agents on a proper subset of servers greater than one. To use this plot, pick a LAN size, for example 4 hosts. We find that moving all hosts to a single server is profitable, in the sense of mean rehydration time, if the probability of server failure is between 0.89 and 1.0.



This is a good time to break now, and get back to programming. It is clear, however, that there are some cases where moving all agents to the same server makes sense, even after accounting for the parallel processing capabilities available in some future multi-threaded agent rehydration solver.

Lim (Pf -> 0) f(Nh) = (Nh * Pf) * (1 - Pf**(Nh-1)) / (1 - (1-Pf)**Nh) < 1

= Lim (fnum') / Lim (fdenom')

The derivative of the numerator is given by:

fnum' = Nh * (1 - Pf**(Nh-1)) + (Nh * Pf) * (- (Nh-1) * Pf**(Nh-2))

fnum' ==> Nh - (Nh*Nh) * Pf **(Nh-1)

fdenom' = - Nh * (1-Pf)**(Nh-1)

so Lim(Pf-> 0) F(Nh, Pf) => [Nh - Nh*Nh*Pf**(Nh-1)] / [- Nh * (1-Pf)**Nh-1)]

==> Lim denum" / Lim denom"

==> Nh*Nh * (Nh-1) * Pf**(Nh-2) / [Nh * (Nh-1) * (1-Pf)**(Nh-2)]

==> Nh * Pf ** (Nh-2) / (1-Pf)**(Nh-2)

==> 0

This is interesting! We have found that it is always preferable to assign agents to a single host in the low failure limit. Let us investigate a specific example, of Nh = 4 servers. According to our calculations, a Pf = 1.0 e-10, and a Pf = .9, will both be preferable in the single server situation. We will leave the case of comparing a parallel threaded rehydration of a system on a single host, to a parallel threaded rehydration on mulitple hosts to the reader, or a later stage of the project. For now, we are getting interesting results that box in our expectations on when to spread agents around, or confine them to a single reliable host.

Recall our earlier results for mean agent rehydration time. We had three cases. A case where all agents are assigned to a single host.

<***Tmove_total***(All agents on one server, single threaded rehydration)> = ***Pf*** * ***Na*** * ***Tmove1*** * (1 - ***Pf*** **(***Nh*** -1))

<*Tmove_total*(agents on all servers, parallel threaded rehydration)> = (*Na* / *Nh*) * *Tmove1* * [1 - (1 - ***Pf***)**_**_***Nh***]

<***Tmove_total***(All agents on one server, single threaded rehydration)> = ***Pf*** * ***Na*** * ***Tmove1*** * (1 - ***Pf*** **(***Nh*** -1))

= (Na * Tmove1) * (1.e-10)/(1-1.e-10)**3 approx= (Na * Tmove1) * 1.e-10

<*Tmove_total*(agents on all servers, parallel threaded rehydration)> = (*Na* / *Nh*) * *Tmove1* * [1 - (1 - ***Pf***)**_**_***Nh***]

= (Na * Tmove1) * (1/4) * (1 - (1-1.e-10)**4) approx= (Na * Tmove1) * (.25)

Therefore, our low failure case of Pf = 1.e-4 should have all agents hosted on the same server.

Now consider the case of Pf = 0.9:

<***Tmove_total***(All agents on one server, single threaded rehydration)> = ***Pf*** * ***Na*** * ***Tmove1*** * (1 - ***Pf*** **(***Nh*** -1))

= (Na * Tmove1) * (0.9)/(1-0.9)**3 = (Na * Tmove1) * 9.e-4

<*Tmove_total*(agents on all servers, parallel threaded rehydration)> = (*Na* / *Nh*) * *Tmove1* * [1 - (1 - ***Pf***)**_**_***Nh***]

= (Na * Tmove1) * (1/4) * (1 - (1-0.9)**4) = (Na * Tmove1) * (.25) * (1 - 1.e-4) = (Na * Tmove1) * .25 * .9999 approx = (Na * Tmove1) * .25

Therefore, we are again in the range where hosting all agents on a single server is preferable, with Pf = 0.9.

Now let's look at an example where we expect this will not be the case, and where we expect a more general distribution of agents to be desirable (with respect to mean agent rehydration time). Consider the case Pf = 0.1.

<***Tmove_total***(All agents on one server, single threaded rehydration)> = ***Pf*** * ***Na*** * ***Tmove1*** * (1 - ***Pf*** **(***Nh*** -1))

= (Na * Tmove1) * (0.1)/(1-0.1)**3 = (Na * Tmove1) * .1 / .9**3 (Na * Tmove1) * (.137)

<*Tmove_total*(agents on all servers, parallel threaded rehydration)> = (*Na* / *Nh*) * *Tmove1*** * [1 - (1 - ***Pf***)**_**Nh**]

= (Na * Tmove1) * (1/4) * (1 - (1-0.1)**4) = (Na * Tmove1) * (.25) * (1 - .9**4) = (Na * Tmove1) * .25 * .3439 = (Na * Tmove1) * .086

Therefore, in the case that we have Pf = 0.1 on a 4 server LAN, we find that it is preferable to move agents onto ALL of the hosts, and not onto a single host.

Conclusions:

In this situation, we introduced a concept that will be referred to as the Mean System Rehydration Time. It is the average amount of time required to rehydrate a system on new servers, following probabilistic failure of the system. Our goal, is to choose a system configuration with the lowest mean rehydration time. This is the second order estimate on optimal agent server assignment, and should help explain the cases where the ex nihilo solver chooses to move all agents to the server with the lowest probability of failure.

It is important to note that the current ex nihilo solver always moves agents to the overall assignment having the lowest probability of failure. It is also important to note that the lowest probability of failure may have higher average times for rehydration compared to other distributions of agents.

This is the enhancement discussed in our Aug. 2002 visit to the TIC. As mentioned during that meeting, the problem of agent rehydration is related to the system reliability problem. Recall that the reliability problem is the sum over all failover states of the system, and is #P-Complete. This is in marked contrast to the NP-Complete problems in other parts of the system.

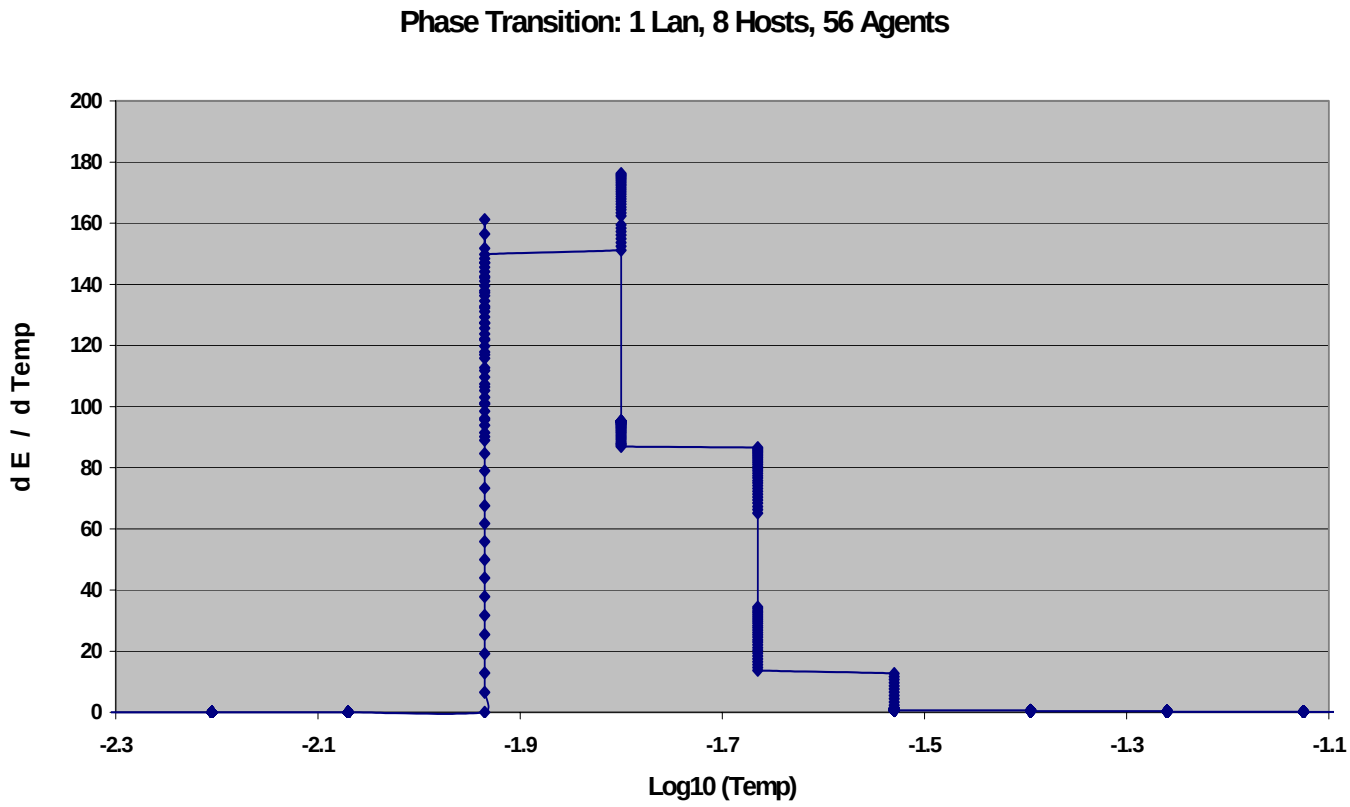
In above discussion, we just compared the situation of moving all agents to a single server, versus moving all agents to all servers, equally distributed, and with equal probabilities of failures. We need to generalize this concept, to find the lowest Mean System Rehydration Time, over all possible agent reassignments, with general probabilities. The Mean System Rehydration functions, similar to the definitions above, will be a major part of the upgrades to the solver. These agent assignments will offer the user the option of moving to the safest possible configuration, or choosing an agent assignment that minimizes the Mean System Rehydration Time. This will be a major upgrade to the solver.

Note that when this question was first raised, I said that it was related to the difference between probability of failure of a single operational state, and the system reliability obtained by integration of all possible failover states. We are now in a good position to build this #P-Complete problem into the solver in a meaningful way.

As a final comment, notice that the Mean Rehydration Time function has an interesting structure. There is a very limited solution range at the low failure limit, followed by a larger solution region in the high failure limit, separated by a medium failure range, where the equal distribution of agents among states is the preferred solution.

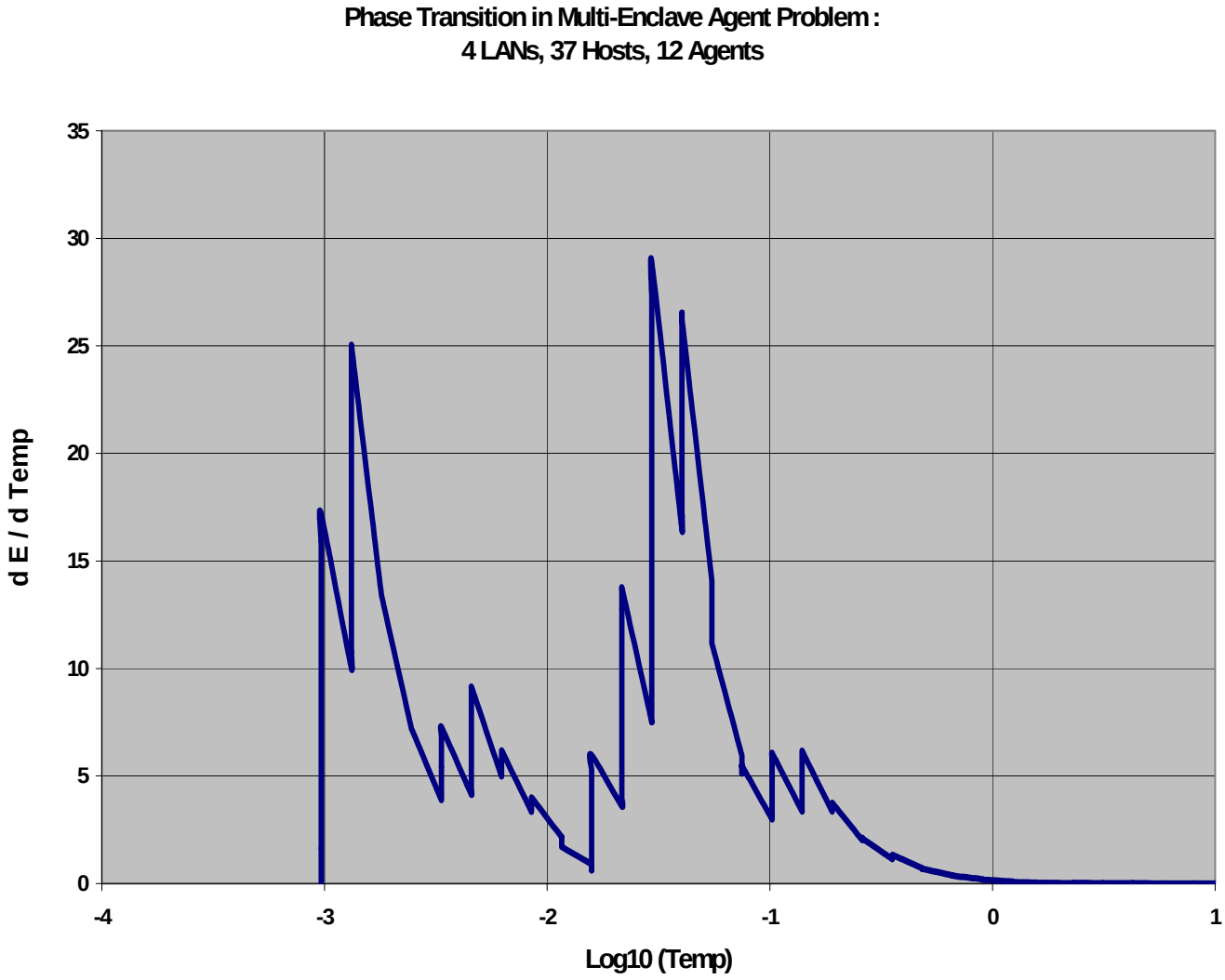
14.) Phase Transitions:

The system exhibits a single phase transition, where small reductions in the temperature produce dramatic improvements in the mean objective function "E". The plot below is a plot of the derivative of the mean energy with respect to temperature. In statistical physics this is known as the "Heat Capacity".



In the above example, there is a clear phase change taking place in the optimizer in the temperature range of 10**-1.5 to 10**-2. By the time the system has frozen into its final solution. We need to use the phase change diagram to automate the annealing schedule. In this 1 LAN, 56 Agent, 8 Host problem, we only see a single phase change taking place.

In the following example of a system with 4 LAN segments, 37 hosts, 12 agents, and complex connectivity, we see a much more interesting example of phase change. There are two clear areas in which the system appears to be undergoing phase transitions. At higher temperature, the system is exploring large scale global structure in the solution, like major routing decisions. At lower temperature, the solution appears to be undergoing a lower order phase transition associated with more fine grained solution details.



The state-of-art solution techniques in simulated are currently focusing on pattern recognition in phase space, where we use phase changes and other entropy measures to signal abrupt changes in the optimization problem. Any further improvements to the automation aspects and speed of solution are expected to have a heavy emphasis on combinatorial optimization viewed as a thermodynamic system.

15.) Suggested Modifications

In this section, I will try to condense some of my thoughts on the proposed future of the ex nihilo solver in Ultralog. There are a large number of proposed modifications, so the following list is a small sample of the major proposals. The color coding is the same as that used on our list of tasks currently being passed around management. I have added new comments in the color Aqua.

Task List

(Legend: High Priority – red; medium – blue; low – green, New - Aqua)

1) Need for Problem input checker. 2-3 weeks.

2) Conversion of Solver and GUI to Another Language [ask Mark about language; for Aug 03 deliverable]

3) Soft constraints [requires resolution of philosophical issue at bottom of list]:

    a. Soft failure of CPU and memory

        i. Allows system to violate hard constraints on CPU and Memory. The solver would search for a solution with the least possible violation of these constraints. It is expected that soft constraint violation may lead to excessive page faults, CPU queuing, or CPU context switching. Continued running in a state that violates hard constraints may lead to system instability.

    b. soft agent mobility groups (coordinated agent moves, move with, not move with)

        i. Some agents would prefer to be on same host as other agents. Some agents to prefer to not be on the same host as other agents.

4) Shared Memory Model

a. In a shared memory model, we would generalize the current agent definition to include effects of shared memory between agents. This would allow for accurate memory modeling in the complex threading present in Ultralog.

5) Factor in the cost of agent movement.

a. Needed to factor in varying rates of damage to a failed system. Some systems with only a few agents on damaged hosts would require less time to rebuild (rehydrate) following damage to system.

b. Inclusion of effects for Mean System Rehydration Time in Agent Assignment problem

6) Solver Speedup (392 hours in recent plan at year's end for speedup operators)

a. Graph homeomorphism vs homomorphism

i. Operator selection should include subsets of major architectural components, like subtrees in system (Homomorphism). Also investigate architectural subcomponents in the operator structure that can be topologically reduced to existing parts of the architecture (homeomorphism).

b. Improved State Space Transition Operators to Speed Convergence

c. Genetic algorithms for detection of nucleation sites, and operator sampling (genetic window integration)

7) Automation of Solver

a. Use of queuing theory limits in determination of Max CPU Utilization. Automatically constraining solver to portions of space that exhibit healthy queuing is expected to reduce overall time to solution.

8) Analytic Seeding (extended speedup of starting solutions, 400 hours in 2003-2004 schedule, "Analytic Seed of Solver")

a. Graph partitioning bootstrap for "response time" solver. Used to minimize messaging, and as a surrogate for response time.

b. Fast multi-dimensional bin packing bootstrap for overall solver, 2 dimensional and 1 dimensional fast check

c. Clustering used to capture of measures of closeness between servers

    i. Consider having parts of space, based on clustering, serve as nucleation sites for the general solver. Reduce temperature for nucleation sites, and pass this information to other parts of system based on entropy.

9) Improved annealing, information theory, entropy, phase transitions, automated annealing schedule (formerly Annealing Schedule Modifications, 700 hours)

a. construct neighborhood structure to smooth out entropy profile

    i. The topology of the space is the one main thing we have under our control in constructing an effective algorithm. We can adjust the topology of the solution space (nearness or closeness of solutions), so that our system has a reasonably smooth entropy profile, and allows easy moves across space in finding global optimum

    ii. Supercooled fluids are obtained when there are no nucleation sites to initiate crystallization. System viscosity increases until system freezes into an amorphous solid, or glass. (no first order phase transition)

10) Move minimization.

a. The current solver will return the state of minimum failure probability regardless of the of improvement over the existing state. For example, if the current state has Pf = 0.1, and the best state has Pf = 0.099999, then the solver will return the optimal state, even though it is an insignificant improvement in the current state, and may require moving every agent in the system.

b. Hamming metric to minimize unnecessary moves

    i. This metric measures the number of state changes from the current state to the proposed state.

c. Minimum improvement in objective function

    i. State changes which differ by small amounts from the current state (like 0.1 versus 0.099999) would be rejected

11) System Reliability Estimates: backup modes, failover plans, and agent reassignments in failure analysis. (This tasks was formerly listed as Pathset Analysis for Failure Analysis) [discuss]

a. Related to Mean System Rehydration Time.

b. Obtained by integrating over all failover modes of system.

c. Problem is in Complexity Class #P-Complete

12) Response Time [RT improvements are problematic due to the lack of appropriate data from the Cougaar side]:

a. Non-Markov Process modeling for response time solver. Needed for reasonable estimates in min and max response times. (title in SOW: Include Analytic Estimates in Support of Non-Markov Processes)

i. Current solver uses analytic M/M/c queuing theory, where exponential workload arrivals and workload service requirements are averaged over all running processes. This would include G/G/c queuing corrections to account for bursty and chaotic nature of work arrivals in the distributed agent environment

b. Update Model Response Times to Reflect Queuing Theory Projections of response on Fully Loaded Solution State of Links

c. Update Model Response Times to Reflect Queuing Theory Projections of response on Fully Loaded Solution State of Nodes

d. Iterate over Paths

e. Inter-enclave agent moves: Corrections for Constraint Limits on Response Time, Failure, Cost, and Monthly Cost

f. MTU Corrections to Routing Algorithm

g. Add Closed Queuing Network Capability to Solver

i. Since the Ultralog system is currently envisioned as a batch type of system, the closed queuing network capability is required.

h. Response Time: Asynchronous function calls [discuss with Zinky]

13) Add or delete agents by on a priority basis (soft failure... some agents fail, some don't)

a. Adjust Unix priorities of agents. (instead of agents turned off, like above, consider giving them a lower priority)

14) Agent/Server Affinity (soft eligibility) [discuss]

a. The current solver allows users to specify server eligibility for a host on a yes/no basis. Either the server S is eligible for execution of agent A, or it is not. This eligibility could be generalized to a continuous weight. For example, we could say that agent A prefers server S1 with weight P(A, S1), and prefers server S2 with weight P(A, S2).

15) Disk modeling [data for this is the real problem]

a. A disk model could be implemented in the same queuing network used in the current solver. This could create more realistic models in situations where there is significant disk IO.

16) Cache modeling [data for this is the real problem]

a. A Cache hit model is possible within the current solver architecture. However, the information requirements to create an effective cache model could be quite demanding.

17) Mirror problem [this is a Cougaar architecture issue]

a. In a mirrored server environment, it would be possible for multiple agents to respond to tasks posted to a blackboard. In the current system, there is one and only one agent that responds to a posted task. By allowing multiple "mirrored copies" of agents to respond, we can improve both system reliability and response time.

18) Possible opportunities for parallel processing techniques, and parallelization of existing serial processes with buried parallel thread capabilities. Possible use of Unix process priorities on a local scale to help global measures of calculation flow in a distributed calculation. Use of multiple nodes per host in system tuning.

a. Multi-thread tracking (was multi-class user modifications). Investigate selected bottleneck threads in solver.

b. Spawning Tasks after Synchronized Closeout of Selected Threads

i. Some task may only be spawned after multiple other conditions have been met, like completion of other threads in the system.

19) Inter-enclave moves (this is a Cougaar operational decision, but EN already handles it). For improved performance:

a. Formal Routing Tables, Smart Routing (State Machine for Routing Matrix)

20) warning form.show:

a. Message severity level: info, failed state, warning, fatal

b. Actions: pause, continue, write to log, post to form, where in code message was generated

21) Enhanced Cost Modeling, Real Options

a. Advanced cost modeling covers both financial modeling, and Real Options applied in an abstract sense to other measures of risk and reward. For example, it may be desirable to "purchase an option" by enforcing low utilization in a candidate host. This low utilization is requested in anticipation of growth of the current thread, given a risky environment of other threads in the system.

22) Parallel (Redundant) Links Between Nodes.

a. The current model only allows a single path of information between two adjacent nodes in the network. Although there may be multiple complex paths, the model does not allow "parallel data links".

23) Marc learning Ultralog, and mounting it at home.

a. I would like to spend quite a bit more time understanding the workings of Cougaar, and the Ultralog application. This is needed to gain insights on new improvements to the distributed agent assignment problem.

24) Data Collection issues:

a. Packet sniffers. www.Ethereal.com

b. Page faults, context switches, and other ps data in /proc

    i. incorporate total % CPU of quiescent period (before send oplan) into background load estimate

c. Realistic latencies (ping/2)

d. Identification of bottleneck threads

e. Estimates of Statistical Agent Resource CPU and Memory Requirements

Philosophical issue on soft constraints:

Say we've got two constraints that conflict such that they can't both be met and still achieve a solution. Is it preferable to return no solution or violate one or other of the constraints? Note that the latter will lead to a requirement that the constraints be prioritized. Is that something that can reasonably be done?

16. Summary

We have covered a lot of ground in this document, and have come a long way since our first meeting in the Feb. 2002 Ultralog Workshop. The current system has been deployed to the test lab (TIC), and is performing well under integration studies.

We began in sections 1 and 2 discussing the difference between the logical and physical topology in a network. Since we are given a logical topology in Ultralog, then routing decisions are not required, so a solver speedup is possible. We also discussed constraint satisfaction in the model, and discussed "satisficing" as a method of simplifying the problem. It was found that some classic statements of "satisficing" are also NP-Complete, and already in the model.

In section 3, we investigated the distributed use of Ex Nihilo, since it is a source of single point failure when used by itself. We also found that the distributed use of solver could be managed by estimating the reliability of the information sources feeding ex nihilo. When network reliability is combined with data time stamps, a distributed ex nihilo solver could be managed by carefully weighing the information sources used by ex nihilo in making its projections.

In Section 4, we extended our discussion of networking, and discussed the use of ex nihilo to put the global multi-enclave problem into approximate block diagonal form. This block diagonal form could be used to simplify inputs to the model, by effectively decoupling LAN segments, and treating the sub-problems as almost independent distributed operating systems.

We also discussed the "No Free Lunch Theorem", which is a useful response to the frequent questions of "why use simulated annealing, and not genetic algorithms?". In general, this theorem states that all combinatorial optimization problems are more or less equal, when averaged over all objective functions. This choice of Simulated Annealing has other motivating factors, such as the ease with which changes to the objective function can be made compared to genetic algorithms. The annealing technique is based on statistical quantum mechanics, compared to the genetic heritage of genetic algorithms. Overall, genetics is easier to explain to the entry level "popular science" community, as opposed to statistical quantum mechanics, and therefore "plays better in the press", since it is at least understandable at a basic level. However, the ease with which an algorithm plays to the popular press should not be a major selection criteria in choice of algorithms.

It is also worth noting that ex nihilo does have a genetic window for capturing interesting pieces of the design space on a probabilistic level. Although ex nihilo uses a simulated annealing backbone for the solver, it does allow the "gene pool" in the model to be sampled for good state space transition operators. Furthermore, the gene pool is found to be an excellent source of information on the physical topology of the system, including backups and failover plans. In this sense, the ex nihilo genetic window is a probabilistic view of the physical topology of the system, while the main window reports outputs in the logical topology.

In Section 6, we discussed CPU, Memory, and Disk modeling, and multi-homing issues in a network. The CPU is modeled as an M/M/c queuing server, averaged over job arrival rates and workload sizes. In the 2002 model, the queuing contributions are turned off, and CPU is modeled as a hard resource constraint. The memory is also modeled as a hard constraint on the servers. The CPU resource requirement for an agent is the "high water mark" of CPU usage during a thread, and is assumed to be a level at which the agent has reasonable acceptable levels of task queuing or time slicing at the CPU. The disks are not modeled in ex nihilo, and therefore needs to be enhanced before studying situations involving significant amounts of disk IO. The system is currently modeled as an open queuing network, and requires conversion to a closed queuing network before we can properly estimate and optimize response times of the running batch system. We also noted in Section 6 that the current solver does not give preference to multi-homed hosts when searching for solutions. A simple weighting for multi-homing could be implemented in the current system in a relatively straightforward fashion, while a more detailed treatment of failover routes and multi-homing would involve a careful investigation of the reliability problem, known to be #P-Complete, since an enumeration of all failover routes is required in a true multi-homing analysis.

In Section 7, we discussed the definition of functions used as inputs to the solver. A good set of functions, that define the CPU and memory usage, messaging, and calling architectures, is the fundamental backbone of any distributed system performance and design optimization problem. Since the function inputs to ex nihilo describe a set of user performance programs that describe system usage, then the input program is seen to bring on problems with the Turing Halting Problem. It is assumed that the users of the tool have checked their program input to ex nihilo, and that the functions so defined by the user reaches a halting state.

In sections 8, 9, and 10, we give a detailed breakdown of the hardware used in the agent assignment problem. Classic quantities like host CPU count, background loads, and memory are required to define a host, while the links are defined in terms of bandwidth, latency, and protocol overheads. There is also a discussion on some of the basic solver parameters used to run the model.

In Section 11, we gave a pedestrian overview of simulated annealing, and described the Metropolis algorithm used in general simulated annealing algorithms. The algorithm is known for its robust character, and is not overly sensitive to changes in the objective function. At high temperature, the solver is allowed to "feel out" the large scale structure of the solution space, while at lower temperature, the algorithm explores the fine grained structure of the space. Since the algorithm can make probabilistic moves that degrade the objective function, then it can escape getting caught in local minima. It is this aspect of the algorithm that makes it effective at global optimization. It can be shown under very broad circumstances that the algorithm converges in distribution to the global optimum, although it requires logarithmic cooling to achieve this distribution. In practice, the simulated annealing algorithm is "quenched", or cooled rapidly to get good solutions in a reasonable amount of time - one of the key strengths of the algorithm.

In Section 12, we reviewed changes in the Ultralog project as a result of the Albuquerque meeting, which was an important event in the project. At the meeting, it was decided that response time modeling was too demanding from a data mining perspective for the 2002 project year. At this meeting, it was decided to focus on "instantaneous rates of resource usages", as opposed to actual resource used by a particular "agent call". For our model, this meant that CPU usage would be measured in units of "CPU seconds per real second per agent", as opposed to the "total number of CPU seconds required by an agent to perform a subtask". Since absolute CPU usage to perform a subtask is an absolute requirement for response time modeling, then the choice of measured units of CPU usage means that response time modeling would be impossible in the short term, and therefore deferred to 2003-2004. We also implemented a memory model that modeled agent usage of unshared memory. For data link utilization, the move to instantaneous rates of resource utilization, meant that our units for data link usage would be "bytes per second", as opposed to the absolute measure of "bytes per message". The Section closed with a discussion of server and data link health, and an explanation of our constraint models.

In Section 13, we gave a highly detailed account of the difference between optimizing a system to minimize probability of failure, and optimizing a system to minimize the "Mean Rehydration Time". The Mean Rehydration Time, was defined as the mean time required to rehydrate a system following partial failure of some of its components. While the current model finds a solution state to the system that minimizes the probability of failure, it is proposed that the model be enhanced to find a state that minimizes the mean time required to rehydrate the system after failure. The Mean Rehydration Time problem is related to the system reliability problem, in that both problems require a summation over possible failover states, and is #P-Complete. The #P-Complete problems are believed to be (in some sense) harder than the NP-Complete problems, since the candidate solutions cannot be verified in Polynomial time (current unsolved problem in mathematics). For this reason, the #P-Complete problems are believed to be outside of the complexity class known as "NP".

A detailed proof was given showing when a set of agents should be moved to a single server to minimize mean system rehydration time, compared to equally spreading the agents over all servers. It was found that this decision on single host versus multi-host assignment had regions in failure space (Pf, probability of server failure) that were quite complex. For some values of Pf, it was favorable to move all agents to a single server, while for other values of Pf, it was favorable to divide all agents equally among all servers.

A generalized solver, based on minimizing Mean System Rehydration Time, would require a careful analysis of the backup modes of the system. The current analysis in this paper is based on homogeneous server failure probabilities. This proposed enhancement to our solver works well with our already proposed reliability enhancements, since the same summation over all backup states of the system is required in both cases.

A generalized model would allow arbitrary failure probabilities, and include link failure contributions. It would also need to consider cases where the summation includes states where the system is given more than the two possible states analyzed in the detailed proof. For example, we would also need to study the situation where agents were divided among 2 servers, 3 servers, and so on, up to Nh-1 servers, where Nh is the number of servers in the system. Recall that our detailed proof only addressed the cases of all agents on 1 server, or all agents divided equally among all Nh servers.

In Section 14, we discussed some of the current phase transition procedures used in combinatorial optimization. When the system of equations in distributed design is treated as a complex thermodynamic system, it reveals a number of interesting things about the solution process.

At distinct points in the cooling history of the problem, there appear a number of regions of phase transition, where the system is altering its microscopic state on a dramatic scale, and producing solutions with dramatic variations in quality on a macroscopic scale. These phase transitions are spotted by looking for major events in the derivative of the mean energy with respect to temperature (known as the heat capacity of the system). Automating and speeding up the annealing solver will rely heavily on the phase change phenomenon seen in the solution process. Basically, we are looking for that point in the system, when the system moves from a "liquid state of agents" to a crystalline state of agents. The freezing or crystallization of the system signals the transition from an unordered high energy state (liquid) to a crystalline state, where (hopefully) the system has self-organized into a highly ordered state as a crystal.

It is worth noting that early investigation of the phase change phenomenon has revealed some interesting results. For a single LAN system of agents, with 56 agents on 8 hosts, there is only a single phase change that has been observed, in the range of 1.0e-2 degrees. In contrast, a complex system with 4 LANs, complex connectivity, 37 hosts, and 12 agents clearly shows two regions of phase change. It appears that the "warmer" phase change is associated with building a solution that discovers the macroscopic features of the architecture, while the cooler phase change is associated with finding solutions in the fine grained structure of the problem.

The last major section of the document presented a large number of suggestions for enhancements to the current solver. Some attempt has been made to prioritize the suggestions for future implementation. The suggestions cover a broad array of technical enhancements associated with response time, failure, search speed, and reliability. Some of the tasks are fairly straightforward bookkeeping, while other tasks like annealing schedule modification and G/G/n queuing theory are expected to be both deep and lengthy.

A number of enhancements have been proposed to "soften" some of the hard constraints in the problem. For example, the model should allow soft violations of CPU and Memory. Other soft enhancements include the ability to model soft agent mobility grouping, and soft agent-server affinity.

A large number of enhancements have been proposed to find fast analytic approximations to a problem, used to seed the simulated annealing solver.

Overall, I think we are on an extremely good plan for the solver, with a large number of excellent suggestions for improvement. With the proposed enhancements, the solver should be able to explore design questions previously thought to be far beyond the capabilities of distributed design analysis, and should provide a solid backbone for future development of distributed agent systems.

References:

[1] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing", Science 220: 671-680, 1983.

[2] Michael R. Garey and David S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*, New York: W.H. Freeman and Company, 1979.

[3] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines*, New York: Wiley, 1989.

[4] David H. Wolpert, William G. Macready, "No Free Lunch Theorems for Search", Feb. 23, 1996. Santa Fe Institute Technical Report, SFI-TR-95-02-010